UNIVERZA V LJUBLJANI

FAKULTETA ZA RAČUNALNIŠTVO IN INFORMATIKO

Grega Tratnik

Razvoj večplatformne mobilne aplikacije s Xamarin.Forms

DIPLOMSKO DELO

VISOKOŠOLSKI STROKOVNI ŠTUDIJSKI PROGRAM PRVE STOPNJE RAČUNALNIŠTVO IN INFORMATIKA

MENTOR: doc. dr. Aleš Smrdel

Ljubljana, 2015

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Tematika naloge:

Vse bolj pomembne postajajo aplikacije za mobilne naprave, kjer prevladujejo tri platforme. Razvoj mobilne aplikacije za čim širši krog uporabnikov tako lahko zahteva razvoj treh (ali tudi več) ločenih aplikacij. Alternativa temu pa je uporaba okolja, ki omogoča razvoj večplatformnih aplikacij. V diplomski nalogi predstavite in testirajte razvojno okolje za izdelavo večplatformnih mobilnih aplikacij *Xamarin.Forms*, ki omogoča hkratno izdelavo izvirnih aplikacij za tri prevladujoče platforme: *Android*, *iOS* in *Windows Phone*. Predstavite elemente in postopek izdelave grafičnega uporabniškega vmesnika ter izdelavo programske kode. Prikažite tudi, kako je mogoče uporabiti posamezne funkcionalnosti, ki jih neka platforma omogoča, druga pa ne. Prikažite tudi zmožnosti za testiranje, ki jih nudi razvojno okolje. Elemente, postopek izdelave in postopek testiranja demonstrirajte na podlagi razvoja in izdelave nekaj mobilnih aplikacij za tri platforme.

IZJAVA O AVTORSTVU DIPLOMSKEGA DELA

Spodaj podpisani Grega Tratnik sem avtor diplomskega dela z naslovom:

Razvoj večplatformne mobilne aplikacije s Xamarin.Forms

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal samostojno pod mentorstvom doc. dr. Aleša Smrdela,
- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko diplomskega dela,
- soglašam z javno objavo elektronske oblike diplomskega dela na svetovnem spletu preko univerzitetnega spletnega arhiva.

V Ljubljani, dne 18. avgusta 2015 Podpis avtorja:

Zahvalil bi se rad družini za podporo in spodbudo tekom študija. Posebna zahvala velja mentorju doc. dr. Alešu Smrdelu za nasvete in nesebično pomoč pri izdelavi diplomskega dela.

Kazalo vsebine

Kazalo tabel

Kazalo slik

Seznam uporabljenih kratic

KRATICA	ANGLEŠKO	SLOVENSKO
LINQ	Language-Integrated Query	jezikovno-integrirana poizvedba
XAML	Extensible Application Markup Language	razširljiv aplikacijski označevalni jezik
XML	Extensible Markup Language	razširljiv označevalni jezik
SAP	Shared Asset Project	projekt v skupni rabi
PCL	Portable Class Library	prenosna razredna knjižnica
MVVM	Model-View-View Model	model-pogled-model pogleda
WPF	Windows Presentation Foundation	predstavitveni temelji za okna
RGB	Red-Green-Blue	rdeča-modra-zelena
HTTP	Hypertext Transfer Protocol	protokol za prenos hiperteksta
MVC	Model-View-Controller	model-pogled-krmilnik
WCF	Windows Communication Foundation	komunikacijski temelji za okna

Povzetek

V današnji informacijski dobi nam modernizacija sledi na vsakem koraku. Internet in pametni telefoni so postali že stalnica in posledično temu narašča uporaba mobilnega interneta, zato si v diplomski nalogi ogledamo enega izmed trenutno naprednejših načinov izdelave mobilnih aplikacij za več platform hkrati – *Xamarin.Forms*. Omogoča nam razvoj aplikacij za platforme *Android*, *iOS* in *Windows Phone*, pri čemer s pomočjo odprtokodnega okolja *Mono* in programskega jezika **C#** dobimo izvorne aplikacije posamezne platforme. V določenih primerih so nam lahko na voljo vse funkcionalnosti, ki jih vsaka platforma ponuja, s čimer nam ta način izdelave ponuja bistveno večjo funkcionalnost v primerjavi z drugimi razvojnimi okolji. V diplomski nalogi predstavimo osnovne elemente okolja *Xamarin.Forms* in prikažemo zmožnosti tega okolja s pomočjo razvoja več praktičnih nalog.

Ključne besede: Xamarin, Xamarin.Forms, Android, iOS, Windows Phone, C#, razvoj večplatformnih aplikacij.

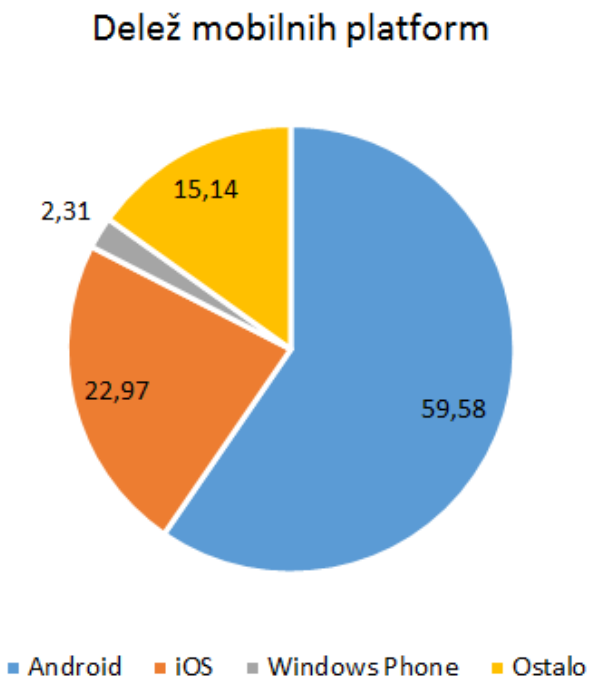
Abstract

In today's informational age modernization follows us at every step. The internet and smartphones have become a constant part of life, and mobile data usage is constantly increasing as a result. In thesis we will examine one of the most advanced current ways of developing cross platform applications at the same time - *Xamarin.Forms*. This enables users to develop applications for *Android*, *iOS* and *Windows Phone* platforms, as an open-source environment *Mono* and programming language **C#** bring us native applications for each platform. In some cases all functionalities per platform can be supported, and *Xamarin.Forms* way of developing applications enables us significantly higher functionality in comparison with other developing environments. In the thesis we will present basic elements of *Xamarin.Forms* with several examples to follow.

Keywords: Xamarin, Xamarin.Forms, Android, iOS, Windows Phone, C#, developing cross platform applications.

1 Uvod

V današnji informacijski dobi se čedalje več internetnih vsebin prenaša na mobilne aplikacije. Z razvojem mobilnih naprav, tablic in ostalih pametnih naprav uporaba mobilnega sveta vidno raste – in trenutno smo šele na začetku cikla [1]. Med najbolj priljubljene operacijske sisteme na mobilnih telefonih spadajo *Android*, *iOS* in *Windows Phone*. Delež mobilnih platform v obdobju junij 2014–junij 2015 je prikazan na sliki 1.1. Vidimo, da med mobilnimi platformami največji delež dosegajo naprave s sistemi *Android*, vendar dosegajo naprave s sistemoma *iOS* in *Windows Phone* nezanemarljiv delež. Zaradi tega je, za dosego čim večjega števila uporabnikov, zaželeno razvijati aplikacije za vse tri platforme.



Slika 1.1. Delež mobilnih platform v obdobju junij 2014–junij 2015.

1.1 Razlike med platformami

Ko se razvijalec odloča za razvijanje mobilnih aplikacij, je veliko odvisno od tega, koliko znanja ima o določeni platformi, saj so razlike med posameznimi platformami lahko zelo velike. Največja izmed njih je predvsem programski jezik, saj se **Objective-C** najbolj razlikuje od **Jave** in **C#** v primeru sintakse. Hkrati obstajajo tudi razlike v razvojnih okoljih in načinu gradnje aplikacij, saj logika vmesnikov in gradnikov med platformami ni enaka.

1.1.1 Razvojna okolja

Uporabnikom so na voljo različna integrirana razvojna okolja za vsak operacijski sistem [2]:

- za razvoj aplikacij *Android* nam je na voljo *Android Studio*, lahko pa uporabimo tudi *Eclipse* oziroma *NetBeans*, za katera pa obstajajo posebni vtičniki,
- pri razvoju aplikacij *iOS* smo žal omejeni na sisteme *Mac*, kjer sta nam na voljo *Xcode* in *AppCode*,
- aplikacije *Windows Phone* lahko razvijamo v *Visual Studiu*.

1.1.2 Programski jeziki

Največje razlike lahko opazimo pri programskih jezikih [4]:

- aplikacije *Android* najlaže razvijamo v **Javi**, obstaja pa tudi možnost razvijanja v **C** in **C++** s pomočjo *Android NDK* [3],
- aplikacije *iOS* razvijamo v **Objective-C** ali **Swiftu**,
- aplikacije *Windows Phone* pa razvijamo v **C#** ali **Visual Basicu,** uporabniški vmesnik pa s *XAML*.

Čeprav so med njimi določene sintaktične razlike, so vsi jeziki objektno orientirani in nasledniki jezika **C**.

1.1.3 Vmesniki in gradniki

S stališča razvijalca prihaja do razlik tudi pri vmesnikih in gradnikih. Slednji so zelo specifični za vsako platformo posebej in v določenih primerih se lahko gradniki obnašajo drugače ter ne podpirajo vseh funkcionalnosti oziroma celo ne obstajajo. Kot primer lahko navedemo vnosno polje, kjer so vidne razlike v poimenovanju na vsaki platformi:

- na platformi *Android* se imenuje *EditText*,
- na platformi *iOS* se imenuje *UITextField*,
- na platformi *Windows Phone* pa *TextBox*.

1.2 Platforma Xamarin

V februarju 2013 so pri podjetju *Xamarin* razvili istoimensko okolje, ki omogoča razvoj izvornih programov za *Android*, *iOS* in *Windows Phone* v programskem jeziku **C#** znotraj okolja *Xamarin Studio* ali *Visual Studio* [5]. Tako *Xamarin.Android* kot *Xamarin.iOS* sta zgrajena na ogrodju odprtokodnega razvojnega okolja *Mono*, ki nam je na voljo kot alternativa tudi na sistemih, na katerih ne teče okolje *.NET*. *Mono* nam je na voljo že od samih začetkov okolja *.NET* in ga lahko uporabljamo na sistemih *Linux*, *Microsoft Windows*, *Mac OS X*, *BSD*, *Sun Solaris*, *Nintendo Wii*, *Sony PlayStation 3*, *Apple iPhone* in *Android* [6]. Uporaba okolja *Mono* kot osnove razvojnega ogrodja omogoča, da lahko s programskim jezikom **C#** razvijamo aplikacije za najbolj priljubljene mobilne platforme. Posebnost *Xamarin.Forms* je tudi v tem, da ne združuje samo avtohtone funkcije, ampak dodaja tudi številne funkcionalnosti:

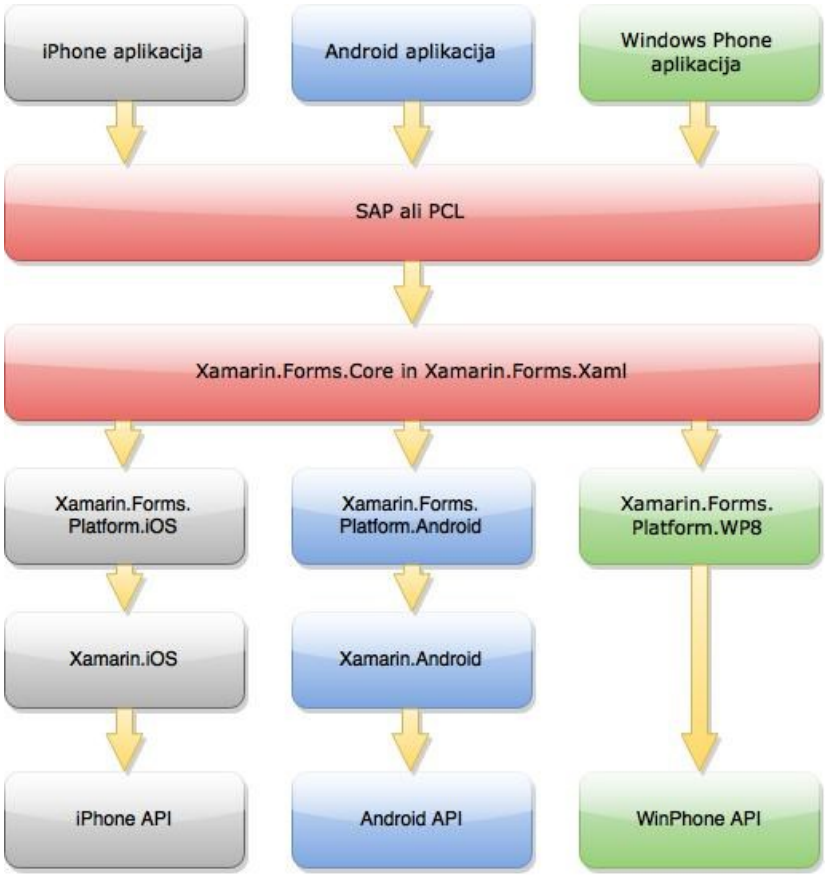
- *Xamarin.Forms* vsebuje popolno preslikavo za skoraj celotno platformo tako za *iOS* kot *Android*,
- omogoča neposredno sklicevanje na knjižnice iz **Objective-C**, **Jave**, **C** in **C++**, s tem pa nam je omogočeno uporabljati široko paleto knjižnic, ki so že ustvarjene,
- napisan je v modernem jeziku **C#**, ki omogoča uporabo lambda izrazov, LINQ, paralelnega programiranja, generikov in mnogo drugih funkcionalnosti, ki jih **Objective-C** in **Java** ne podpirata.

1.3 Xamarin.Forms

Xamarin.Forms je bil predstavljen 28. 5. 2014 kot kolekcija izboljšav za okolje *Xamarin* [7]. Omogoča nam razvoj uporabniških vmesnikov v jeziku *XAML*, ki so potem prevedeni za vsako platformo posebej. Ena izmed večjih pomanjkljivosti trenutno je, da ni na voljo grafičnega urejevalnika za *XAML*. S tem smo prepuščeni razvijalčevi sposobnosti umeščanja gradnikov v format *XML*. Projekt *Xamarin.Forms* je sestavljen iz štirih podprojektov:

- projekt *Android*,
- projekt *iOS*,
- projekt *Windows Phone*,
- skupna koda.

Levji delež kode predstavlja skupna koda, preostali pa se le sklicujejo na njo. Slika 1.2 prikazuje shematsko strukturo projekta *Xamarin.Forms*.



Slika 1.2. Struktura projekta Xamarin.Forms.

Kot je razvidno iz diagrama na sliki 1.2, imamo pri razvoju dve možnosti. Lahko se odločimo za *Shared Asset Project (SAP)* ali pa za *Portable Class Library (PCL)*.

1.3.1 Shared Asset Project

Pri SAP lahko izvorno kodo delimo med projekti in v primeru, da želimo znotraj projekta imeti specifičen klic na točno določeno platformo, bi v primeru za *Android* morali uporabiti direktivo:

```
#if __ANDROID__
        deviceLabel.Text = "Dobrodošli na Android napravi!";
#endif
```

Slabost v tem primeru je, da se poslabša berljivost kode, hkrati pa ni možna uporaba teh direktiv v jeziku *XAML*. Največja težava pri uporabi načina SAP je testiranje, saj ločeno testiranje projektov ni možno.

1.3.2 Portable Class Library

Kadar uporabimo PCL, se med projekti deli binarna koda, s čimer dobimo večjo fleksibilnost, saj jo lahko poljubno uporabljamo tudi v ostalih aplikacijah ali projektih [8]. Hkrati lahko znotraj projekta uporabimo vse knjižnice, ki so bile razvite za ta način. V primeru, da izvajamo specifičen klic na določeno platformo, uporabimo razred *Device*, ki ga lahko kličemo tako znotraj izvorne kode kot v datoteki *XAML*. V praksi se največkrat uporablja slednji način, saj nam poleg naštetega omogoča tudi pisanje testnih primerov za ločene module.

2 Elementi Xamarin.Forms

V tem poglavju si bomo ogledali osnovne gradnike platforme *Xamarin.Forms*. Prikazali bomo, kako gradimo uporabniški vmesnik s pomočjo razširljivega označevalnega jezika, delovanje gradnikov, dogodkov in način deljenja programske kode.

2.1 XAML

Pri načrtovanju grafičnega vmesnika imamo dve možnosti. Lahko ga napišemo kot del programske kode ali pa se odločimo za tehnologijo, ki temelji na jeziku *XML – XAML*. Jezik *XAML* so razvili pri Microsoftu in je le nekaj let mlajši od jezika **C#** [9]. *XAML* je alternativa programiranju kode in nam omogoča deklariranje objektov znotraj jezika *XML* in organizacijo teh objektov v hierarhiji starš-otrok. Trenutno se uporablja v številnih tehnologijah *.NET*, kot so *Windows Presentation Foundation (WPF)*, *Silverlight* in *Windows Phone* [10]. Ker obstajajo za vsako platformo določene posebnosti, so morali *XAML* prirediti za vsako platformo posebej, zatorej med njimi obstajajo tudi določene razlike. Trenutno še tudi ni na voljo grafičnega urejevalnika *XAML* za *Xamarin.Forms*, kar pomeni, da razvijalec nima vpogleda v dejanski izgled aplikacije pred njeno izgradnjo. Znotraj jezika lahko uporabnik definira poglede, razvrščevalnike, strani in tudi lastne razrede. Dobre strani uporabe *XAML* so:

- je bolje berljiv v primerjavi z ekvivalentno kodo,
- pišemo ga lahko ročno,
- zaradi hierarhije starš-otrok si laže predstavljamo strukturo vmesnika,
- enostavneje ga lahko urejamo s programskim vmesnikom kakor samo kodo.

Kot primer ustvarimo vnosno polje (*Entry*) z imenom *entry_Password* in določenimi lastnostmi.

Znotraj kode bi ga ustvarili na naslednji način:

```
var entry_Password = new Entry
{
   VerticalOptions = LayoutOptions.CenterAndExpand,
   IsEnabled = false,
   IsPassword = true,
   Placeholder = "Vnesi geslo",
   Scale = 1.5,
   Color = Color.Blue
};
```

V načinu *XAML* pa tako:

```
<Entry x:Name="entry_Password"
      VerticalOptions="CenterAndExpand"
      IsEnabled="False"
      IsPassword="True"
      Placeholder="Vnesi geslo"
      Scale="1.5"
      Color="Blue"/>
```

Kot je razvidno iz zgornjega primera, se razredi, kot so *Entry* in *Label*, preslikajo v elemente *XML*, lastnosti, kot so *VerticalOptions*, *IsEnabled* in *IsPassword*, pa postanejo lastnosti *XML* znotraj *XAML*. Da lahko elemente deklariramo v *XAML,* morajo slednji imeti javen neparametričen konstruktor, lastnosti pa morajo imeti javne metode za nastavljanje (»setterje«). Postavi pa se vprašanje, na kakšen način bo *Xamarin* pretvoril niz *Color.Blue* znotraj *XAML* v pripadajočo vrednost. Tukaj nastopi refleksija. Aplikacija bo z njeno pomočjo ugotovila, ali so te lastnosti v tem primeru tipa *Color*. Razred *Color* vsebuje lastnost *TypeConverter*:

```
[TypeConverter (typeof(ColorTypeConverter))]
public struct Color {}
```

Lastnost *TypeConverter* nad razredom *Color* se sklicuje na razred, imenovan *ColorTypeConverter*. Ta je zasebni razred, gledano z vidika *Xamarin.Forms*, a deduje iz javnega abstraktnega razreda, imenovanega *TypeConverter,* ki pa vsebuje metodi, imenovani *CanConvertFrom* in *ConvertFrom*. Na takšen način bo prevajalnik lahko dobil vrednost iz niza in ga pretvoril v pripadajoči objekt.

2.1.1 Posebnosti XML

Ko začnemo pisati elemente uporabniškega vmesnika znotraj *XAML*, kaj hitro ugotovimo, da obstajajo določene posebnosti, ki niso samoumevne za jezik *XML* – a hkrati tudi ne kršijo pravil. Namesto da nekaj zapišemo kot lastnost, lahko to v jeziku *XML* zapišemo tudi kot element. Naslednji lastnosti lahko zapišemo na dva načina:

```
<Frame HorizontalOptions="CenterAndExpand"
       VerticalOptions="FillAndExpand"
       IsEnabled="True"
       Content="Vnesite uporabniško ime:"/>
```

Ali pa kot elemente:

```
<Frame>
 <Frame.HorizontalOptions>
  CenterAndExpand
 </Frame.HorizontalOptions>
 <Frame.VerticalOptions>
  FillAndExpand
 </Frame.VerticalOptions>
 <Frame.Content>
  <Label Text="Vnesite uporabniško ime:"></Label>
 </Frame.Content>
 <Frame.IsEnabled>
  True
 </Frame.IsEnabled>
</Frame>
```

Prva izmed razlik, ki jih takoj opazimo, je, da smo vrednosti lastnosti *Content* v drugem primeru dodali celoten objekt *Label*. Hkrati smo lastnosti definirali tako, da smo dodali še ime starševskega objekta in ju ločili s piko. S tem so vpeljana določena poimenovanja:

- *Frame* in *Label* sta v tem primeru objekta **C#,** predstavljena kot elementa *XML* – imenujemo ju objektna elementa,
- lastnosti **C#,** kot so *HorizontalOptions*, *VerticalOptions* in *IsEnabled*, so predstavljeni z lastnostmi *XML* – lastnostmi atributov,
- *Frame.Content* je lastnost **C#,** predstavljena kot element *XML* – lastnost elementa.

2.1.2 Specifični klici za platforme

Pri razvijanju mobilne aplikacije, ki jo želimo podpirati na več platformah hkrati, hitro ugotovimo, da se platforme med seboj razlikujejo do te mere, da niso vse funkcije podprte na vseh platformah oziroma se razlikuje njihovo delovanje. Zaradi tega lahko znotraj *XAML* definiramo specifične klice za določeno platformo. Pri trenutnih nastavitvah se prikaz na telefonih *iOS* prične nad orodno vrstico in moramo dodati zapolnitev (*Padding*) le v primeru, ko ima uporabnik napravo *iOS* [11]:

```
<ContentPage.Padding>
        <OnPlatform x:TypeArguments="Thickness"
         iOS="0, 20, 0, 0"
        Android="0, 0, 0, 0"
         WinPhone="0, 0, 0, 0" />
</ContentPage.Padding>
```

Razred *OnPlatform* je generičnega tipa in definira spremembo sebe do generičnega argumenta – posledično mu lahko kot argument podamo množico drugih objektov.

2.1.3 Dostopi do konstruktorjev in statičnih metod

Kot smo že omenili, lahko v datoteki *XAML* definiramo samo elemente, ki imajo neparametrične konstruktorje. A hitro se zgodi, da potrebujemo elemente s parametri. Za te primere sta definirana elementa *x:Arguments* in *x:FactoryMethod*. Ampak kaj storiti, če imamo element, ki sprejema samo število tipa *Decimal*? Za te primere *XAML* podpira tudi osnovne tipe, definirane s predpono *x:*, npr. *x:Boolean, x:Int32, x:String, x:DateTime*. Pogoj, da bo prevajalnik uspešno prevedel kodo, je, da je število elementov znotraj *x:Arguments* enako številu enega izmed konstruktorjev. RGB-barvo teksta definiramo na naslednji način:

```
<Entry Text="Vnesi geslo">
 <Entry.TextColor >
  <Color>
   <x:Arguments>
    <x:Double>215</x:Double>
    <x:Double>40</x:Double>
    <x:Double>40</x:Double>
   </x:Arguments>
  </Color>
 </Entry.TextColor>
</Entry>
```

Z *x:FactoryMethod* imamo možnost klicati statične metode. Istemu elementu tokrat definiramo barvo iz statične metode *FromRgba*:

```
<Entry Text="Vnesi geslo">
 <Entry.TextColor >
  <Color x:FactoryMethod="FromRgba">
   <x:Arguments>
    <x:Double>215</x:Double>
    <x:Double>40</x:Double>
    <x:Double>40</x:Double>
    <x:Double>0.9</x:Double>
   </x:Arguments>
  </Color>
 </Entry.TextColor>
</Entry>
```

2.1.4 Dostop do elementov XAML v kodi

Do elementov *XAML* lahko v kodi dostopamo na več načinov. Lahko se sprehodimo po drevesni strukturi in iščemo iskani element ali pa elementu dodamo lastnost *x:Name.* Pravila za naslavljanje imen so enaka kot za imena spremenljivk **C#**. Ime se mora pričeti s črko ali podčrtajem in sme vsebovati zgolj črke, podčrtaje in številke. Hkrati lahko obstaja le en primerek imena, v nasprotnem primeru se koda ne bo prevedla. Denimo, da moramo ob zagonu aplikacije vedno nastaviti datum v elementu *DatePicker* na prejšnji teden. Najprej moramo nastaviti ime v datoteki *XAML*:

```
<DatePicker x:Name="_dateFrom"

            Format="yyyy-MM-dd"/>
```

Do elementa lahko v kodi dostopamo na naslednji način:

```
var dateFrom = this.FindByName<DateTime>("_dateFrom");
dateFrom = DateTime.Today.AddDays(-7);
```

Prevajalnik *XAML* se sprehodi skozi datoteko in vsaka lastnost *x:Name* postane zasebno polje znotraj generirane kode [12]. Sprva so vrednosti teh polj *null*, le po klicu *InitializeComponent* se vrednosti polj nastavijo prek metode *FindByName*, ki je definirana znotraj razreda *NameScopeExtensions*. S tem se do elementov lahko dostopa enako, kot bi bili deklarirani znotraj kode. Ta polja so zasebna, zato so dostopna samo iz trenutne datoteke in ne iz ostalih razredov.

2.1.5 Podatkovna vezava

Pri nastavljanju dinamičnih vrednosti lastnosti elementom se vprašamo, ali morda obstaja tudi drug način poleg neposrednega dostopa do elementa. Pri tem se opremo na tako imenovano podatkovno vezavo (*Data Binding*). Slednja omogoča avtomatsko nastavljanje dogodkov in prenašanje informacij med lastnostmi, tako da nam ni treba spreminjati lastnosti neposredno nad elementi. Za to povezavo v večji meri skrbi vmesnik *INotifyPropertyChanged*. *INotifyPropertyChanged* vsebuje dogodek *PropertyChanged*, ta pa nam pove, ali se je vir podatka spremenil, in v primeru spremembe se bo ciljna lastnost spremenila [13]. Takšen način spreminjanja vrednosti lastnostim je značilen predvsem za arhitekturni slog model-pogled-model pogleda (»Model-View-View Model«, MVVM).

Recimo, da želimo rotirati sliko na podlagi vrednosti, ki jo lahko spreminjamo z elementom *Stepper*. Pri tem omejimo minimalno vrednost na 0 in maksimalno na 360. Hkrati bomo še dodali vnosno polje ter njeno vrednost prepisali v oznako. Prvi primer bo narejen zgolj z *XAML*, drugi pa v kombinaciji *XAML* in kode.

Prvi primer naredimo na naslednji način:

```
<StackLayout>
  <StackLayout.Children>
    <Image Source="logo.png" BindingContext="{x:Reference stepper}" Rotation="{Binding Value}"></Image>
    <Stepper Minimum="0" Maximum="360" Increment="10" x:Name="stepper" HorizontalOptions="CenterAndExpand"/>
    <Label BindingContext="{x:Reference stepper}" Text="{Binding Value, StringFormat='Trenutna vrednost rotacije: {0} ° stopinj.'}" HorizontalOptions="CenterAndExpand"></Label>
  </StackLayout.Children>
</StackLayout>
```

Kot lahko vidimo iz zgornje kode, smo morali na sliki definirati lastnost *BindingContext*, ki nam pove, na kateri kontekst se nanaša – v tem primeru na objekt z imenom *stepper*. Ker je vrednost, ki nas zanima, shranjena v lastnosti *Value*, jo podamo v lastnost *Rotation*. S tem se bodo definirali vsi potrebni dogodki in rotacija slike bo potekala glede na vrednost, ki jo podamo v objektu z imenom *stepper*.

Pri drugem primeru moramo definirati več stvari. Najprej ustvarimo nov razred z imenom *Model*. Ta mora dedovati iz razreda *INotifyPropertyChanged*, kjer definiramo dogodek *PropertyChanged,* metodo *NotifyPropertyChanged* in lastnost *UserEntry*.

```
class Model : INotifyPropertyChanged
{
  public event PropertyChangedEventHandler PropertyChanged;
  protected virtual void NotifyPropertyChanged(string propertyName)
```

```
    {
        if(PropertyChanged != null)
            PropertyChanged(this, new PropertyChangedEventArgs(propertyName));
    }
    private string _userEntry;
    public string UserEntry
    {
        get
        {
            return _userEntry;
        }
        set
        {
            _userEntry = value;
            NotifyPropertyChanged("UserEntry");
        }
    }
}
```

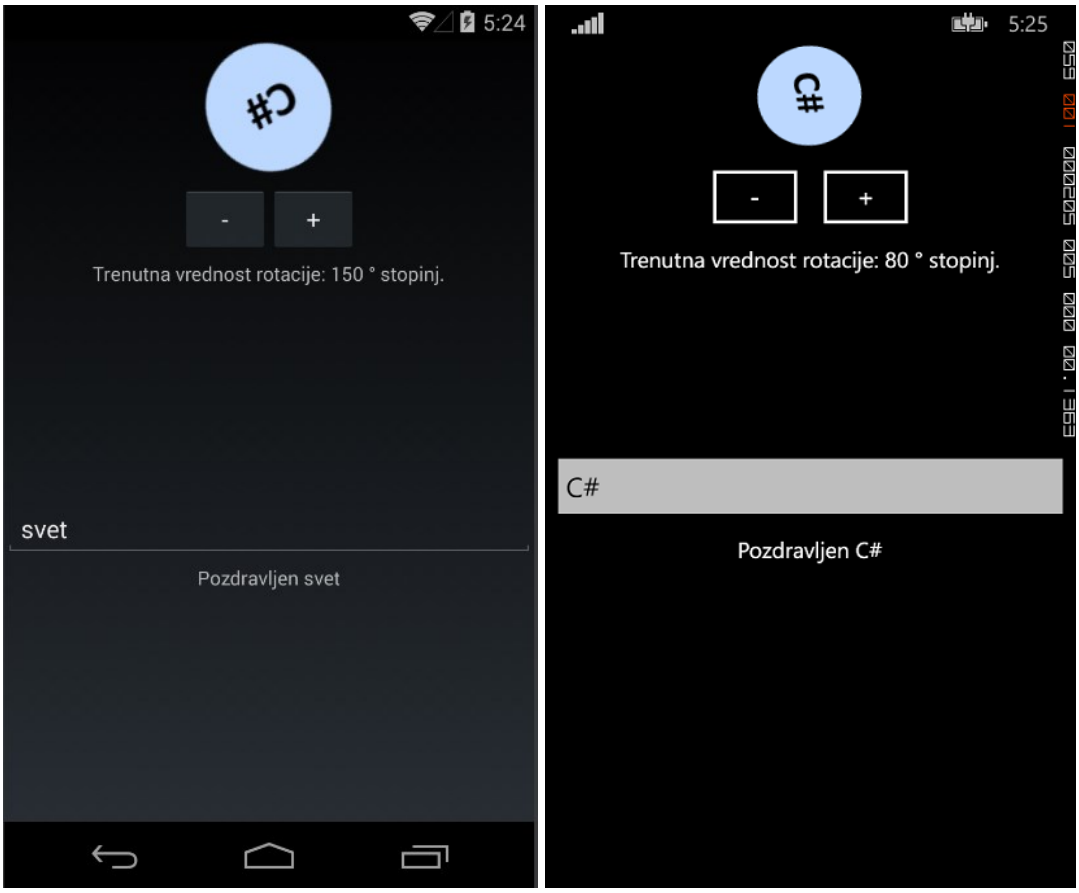
Da bo aplikacija vedela, na kateri kontekst mora biti slednji pogled vezan, moramo to definirati v samem pogledu.

```
public partial class MainPage : ContentPage
{
    public MainPage()
    {
        InitializeComponent();
        this.BindingContext = new Model();
    }
}
```

Sedaj nam manjka samo še definicija znotraj datoteke *XAML*, kjer v vnosnem polju in oznaki nastavimo referenco na lastnost *Username*.

```
<StackLayout Padding="0, 150, 0,0">
  <StackLayout.Children>
    <Entry Placeholder="Vnesi ime" Text="{Binding Username}"></Entry>
    <Label Text="{Binding Username,StringFormat='Pozdravljen {0}'}" HorizontalOptions="CenterAndExpand"></Label>
  </StackLayout.Children>
</StackLayout>
```

Rezultat obeh primerov za platformo *Android* in za platformo *iOS* je prikazan na sliki 2.1.



Slika 2.1. Podatkovna vezava na napravah Android (levo) in Windows Phone (desno).

2.2 Dogodki

V *Xamarin.Forms* imamo več načinov, kako zaznati uporabnikov dotik določenega elementa. Največkrat smo v interakciji z gumbi (*Button*), zato je najenostavneje registrirati dogodek *Clicked*. To storimo na enak način, kot če bi dodajali določeno vrednost lastnostim znotraj datoteke *XAML*, s tem da mora metoda nato obstajati še v kodi.

```
<Button x:Name="btn_Prijava"
                    Text="Prijavi me"
                    Clicked="ButtonPrijava_OnClicked" />
```

Istoimensko metodo nato definiramo še v kodi:

```
public void ButtonPrijava_OnClicked(object sender, EventArgs args)
{
   var isValid = ValidateUserCredentials(username, password);
   if (isValid)
   {
      ...
   }
}
```

Če želimo, lahko vse to storimo tudi znotraj kode, a moramo imeti referenco na želeni element (vedno definiramo lastnost *x:Name*).

```
this.btn_Prijava.Clicked += Btn_Prijava_Clicked;
```

Kaj storiti v primeru, ko želimo zaznati uporabnikov dotik ostalih elementov? *Xamarin* je to zadevo rešil tako, da vsi razredi, ki dedujejo od razreda *View*, zaznajo te dotike prek lastnosti, imenovane *GestureRecognizers*. Trenutno je na voljo razred *TapGestureRecognizer,* ki pokriva iste funkcionalnosti kot dogodek *Clicked*, vendar imamo omogočene še dodatne lastnosti [14]. Denimo, da želimo imeti v aplikaciji omogočen dvojni klik:

```
<Button x:Name="btn_Prijava" Text="Prijavi me">
 <Button.GestureRecognizers>
    <TapGestureRecognizer NumberOfTapsRequired="2"        Tapped="TapGestureRecognizer_OnTapped"/>
</Button.GestureRecognizers>
</Button>
```

Tako kot pri dogodku *Clicked* moramo tudi tukaj definirati metodo v kodi, oziroma če želimo, lahko ves ta blok v datoteki *XAML* nadomestimo z definicijo znotraj kode:

```
var recognizer = new TapGestureRecognizer();
recognizer.NumberOfTapsRequired = 2;
recognizer.Tapped += (sender, args) => {
   .....
};
btn_Prijava.GestureRecognizers.Add(recognizer);
```

Zastavlja se nam vprašanje, katerega tipa je objekt, v tem primeru spremenljivka *sender*. Ne glede na to, da je pošiljatelj v večini primerov objekt, ki ga sproži, je *TapGestureRecognizer* definiran tako, da bo pošiljatelj vedno objekt, ki ga kliknemo.

Obstaja še tretji način definiranja dogodkov, in sicer prek lastnosti *Command* in *CommandParameter*, kar bo posebej obrazloženo v poglavju, namenjenemu arhitekturnemu slogu MVVM.

2.3 Gradniki

V *Xamarin.Forms* obstajajo štiri skupine gradnikov, s katerimi gradimo uporabniški vmesnik, imenovan vizualni elementi [15]:

- stran *(Page)*,
- razvrščevalnik *(Layout)*,
- pogled *(View)*,
- celica *(Cell)*.

Ob izgradnji aplikacije se bodo vse kontrole preslikale v elemente, odvisno glede na operacijski sistem, kjer se bo aplikacija izvaja.

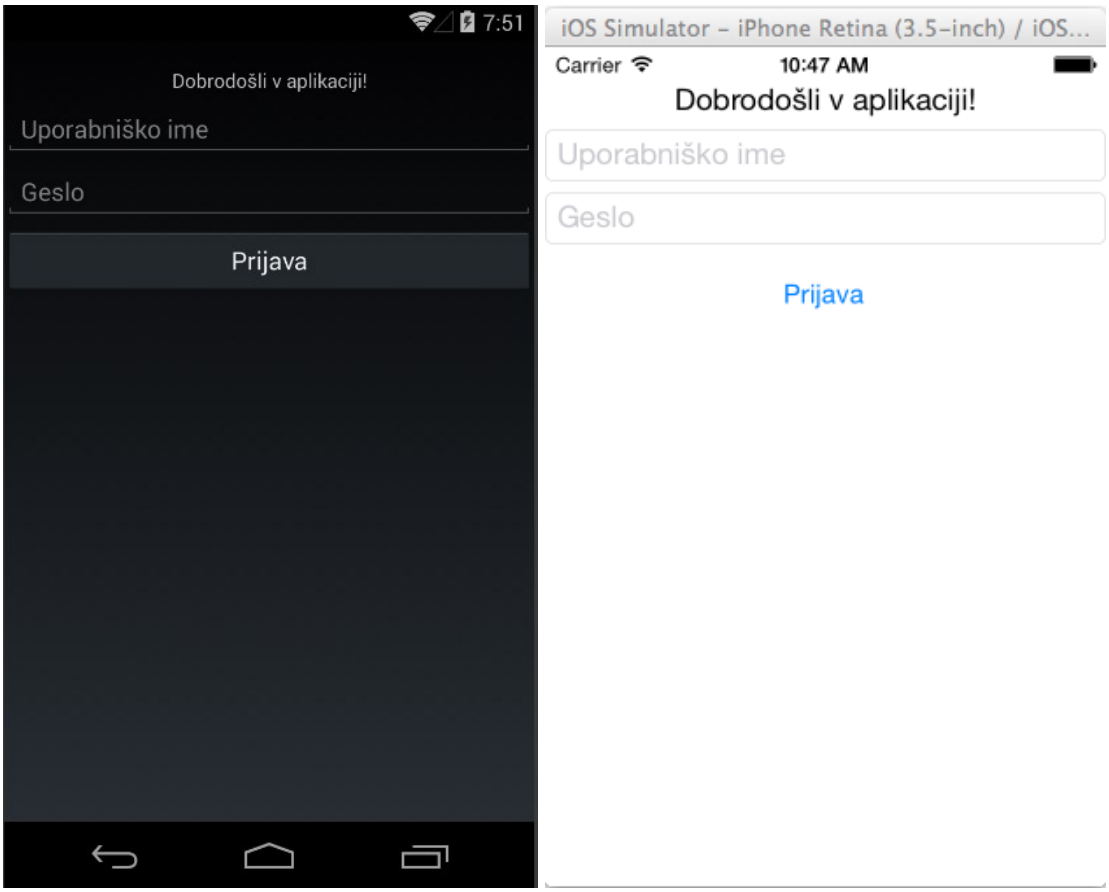
2.3.1 Strani

Aplikacija je sestavljena iz ene ali več strani in predstavlja trenutno okno v aplikaciji. Stran predstavlja *Activity* v aplikaciji *Android*, *ViewController* v *iOS* in *Page* v *Windows Phone*. Vse strani se preslikajo v izvorne strani glede na operacijski sistem. Poznamo jih več [16]:

- stran za vsebino (*ContentPage)*,
- glavna – detajlna stran (*MasterDetailPage*),
- navigacijska stran (*NavigationPage*),
- stran z zavihki (*TabbedPage*),
- vrtiljak strani (*CarouselPage*).

2.3.1.1 Stran za vsebino

ContentPage je najosnovnejša stran v *Xamarin.Forms*. Predstavlja pogled, kamor dodamo poljubne razvrščevalnike, in je hkrati tudi največkrat uporabljena pri kreiranju aplikacij. Prijavno okno z razvrščevalnikom *StackLayout* za vse tri podprte platforme je prikazano na slikah 2.2 in 2.3.



Slika 2.2. ContentPage na napravah Android (levo) in iOS (desno).

Slika 2.3. ContentPage na napravi Windows Phone.

2.3.1.2 Glavna – detajlna stran

MasterDetailPage stran, ki omogoča prikazovanje dveh strani hkrati, se deli na*:*

- višji nivo podatkov (*Master*) – največkrat se uporabi za meni,
- nižji nivo podatkov (*Detail*) – prikazuje informacije glede na izbrano opcijo v strani *Master*.

2.3.1.3 Navigacijska stran

NavigationPage se uporablja predvsem, kadar imamo aplikacije, pri katerih prikazujemo več strani in želimo omogočiti sprehod na prejšnje strani. Deluje po principu sklada, kjer lahko strani enostavno dodajamo in brišemo [17].

Po kliku na gumb izbrišemo trenutno stran in dodamo novo:

```
public async void ButtonPrijava_OnClicked(object sender, EventArgs args)
{
      await Navigation.PopAsync();
      await Navigation.PushAsync(new DataPage())});
}
```

Kot parameter sprejema razred *Navigation* objekte tipa *Page*, kar nam omogoča dodajanje vseh strani.

2.3.1.4 Stran z zavihki

TabbedPage predstavlja kolekcijo strani, podobno kot *NavigationPage*. Razlika je v tem, da imamo namesto navigacije predstavljeno kolekcijo zavihkov. K naslovu lahko dodamo še ikono, vendar ni podprta na vseh platformah.

Primer generiranja datoteke *XAML* z dvema zavihkoma:

```
<TabbedPage xmlns="http://xamarin.com/schemas/2014/forms"
        xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
        x:Class="App7.MainPage">
 <TabbedPage.Children>
 <ContentPage Title="Timeseries">
  <ContentPage.Content>
    ...
 </ContentPage>
 <ContentPage Title="RSS">
  <ContentPage.Content>
    ....
  </ContentPage.Content>
 </ContentPage>
</TabbedPage.Children>
</TabbedPage>
```

2.3.1.5 Vrtiljak strani

CarouselPage je glede na funkcionalnost in definicijo skoraj enak kot *TabbedPage*. Razlikuje se v načinu preklapljanja med stranmi, saj se med zavihki prestavljamo s potegom po zaslonu. Deluje po enakem principu kot galerija slik.

2.3.2 Razvrščevalniki

Razvrščevalniki (*Layouts*) v *Xamarin.Forms* so namenjeni organizaciji kontrol na straneh – vsebujejo napredno logiko, ki omogoča nastavljanje pozicije in velikosti elementov znotraj aplikacije [18]. Z različnimi razvrščevalniki lahko dosežemo enak učinek, vse je odvisno le od načina, kako želimo to izpeljati. Trenutno jih obstaja sedem:

- razvrščanje po skladu (*StackLayout*),
- absolutno razvrščanje (*AbsoluteLayout*),
- relativno razvrščanje (*RelativeLayout*),
- mrežno razvrščanje (*GridLayout*),
- pogled vsebine (*ContentView*),
- razvrščanje z drsniki (*ScrollView*),
- okvir (*Frame*).

2.3.2.1 Razvrščanje po skladu

Eden izmed lažjih načinov organizacije kontrol na strani nam omogoča *StackLayout*. Kontrole se dodajajo sekvenčno z uporabo sklada. Lastnost *Orientation* določa, ali se bodo kontrole prikazovale vertikalno (privzeta vrednost) ali horizontalno [19]. Pomembna lastnost pri izbiri tega razvrščevalnika je, da se velikost in širina kontrol avtomatsko prilagodita razpoložljivim vrednostim.

2.3.2.2 Absolutno razvrščanje

Če se pri postavitvi kontrol odločimo za *AbsoluteLayout*, bomo v mislih imeli najverjetneje bolj kompleksno postavitev pogleda. Ta način predstavlja naprednejši razvrščevalnik, ki omogoča vizualno zahtevnejše postavitve, saj je treba nastaviti koordinate za pozicije kontrol. Ključni lastnosti sta [20]:

- *LayoutFlags* – določa, ali se bodo kontrole porazdelile sorazmerno ali pa v specifičnih enotah,
- *LayoutBounds* – nastavi pozicijo, višino in širino kontrole.

Čeprav nastavitev pozicije predstavlja večji izziv, nam *AbsoluteLayout* omogoča fleksibilnost, saj lahko s pravilnimi nastavitvami dosežemo enak razpored na različnih napravah in dimenzijah. Urejamo lahko tudi prekrivanje kontrol.

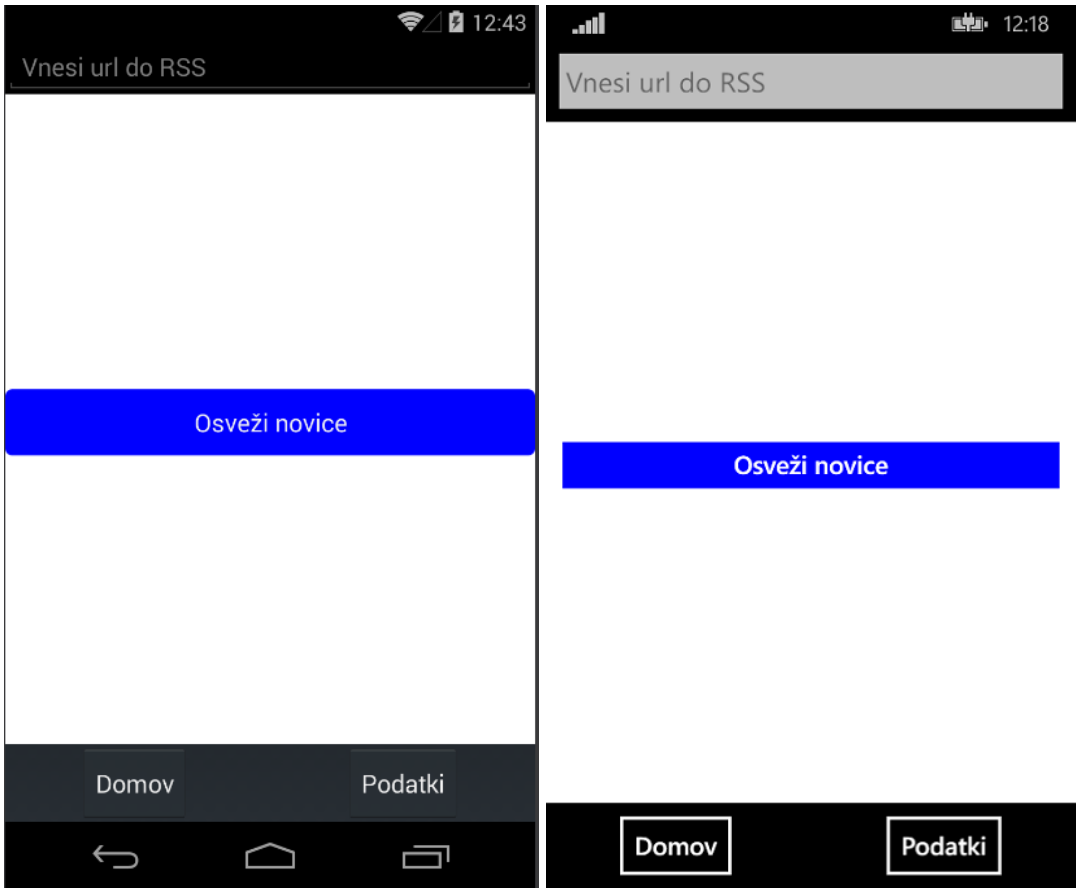
2.3.2.3 Relativno razvrščanje

RelativeLayout je tip razvrščevalnika, kjer se uporabljajo omejitve pri razvrščanju kontrol znotraj strani [21]. Določamo lahko omejitve za višino, širino, X-koordinato in Y-koordinato za kontrolo, ki so relativne v razmerju z ostalimi kontrolami. Pomembna lastnost pri uveljavljanju omejitev je *Factor*, s katerim lahko procentualno določimo razmerje do določene kontrole.

Spodnja koda prikazuje, kako narediti orodno vrstico na dnu strani, ki zaseda 10 % višine.

```
<RelativeLayout>
 <RelativeLayout.Children>
  <StackLayout RelativeLayout.HeightConstraint ="{ConstraintExpression Type=RelativeToParent, Property=Height,
Factor=0.9}"    RelativeLayout.WidthConstraint="{ConstraintExpression Type=RelativeToParent, Property=Width,
Factor=1}">
    <StackLayout.BackgroundColor>
     <Color x:FactoryMethod="FromRgba">
      <x:Arguments>
       <x:Double>22</x:Double>
       <x:Double>73</x:Double>
       <x:Double>154</x:Double>
       <x:Double>255</x:Double>
      </x:Arguments>
     </Color>
    </StackLayout.BackgroundColor>
    <StackLayout.Children>
     <Entry Placeholder="Vnesi url do RSS" BackgroundColor="Black" ></Entry>
     <Button Text="Osveži novice" BackgroundColor="Blue" VerticalOptions="CenterAndExpand"></Button>
    </StackLayout.Children>
  </StackLayout>
  <StackLayout Orientation="Horizontal" RelativeLayout.HeightConstraint ="{ConstraintExpression
Type=RelativeToParent, Property=Height, Factor=0.1}" RelativeLayout.YConstraint= "{ConstraintExpression
Type=RelativeToParent, Property=Height, Factor=0.9}" RelativeLayout.WidthConstraint="{ConstraintExpression
Type=RelativeToParent, Property=Width, Factor=1}">
    <StackLayout.Children>
     <Button Text="Domov" HorizontalOptions="CenterAndExpand"></Button>
     <Button Text="Podatki" HorizontalOptions="CenterAndExpand"></Button>
    </StackLayout.Children>
  </StackLayout>
 </RelativeLayout.Children>
</RelativeLayout>
```

Rezultat kode na platformah *Android* in *Windows Phone* je prikazan na sliki 2.4.



Slika 2.4. RelativeLayout na napravah Android (levo) in Windows Phone (desno).

2.3.2.4 Mrežno razvrščanje

V *Xamarin.Forms* obstaja tudi razvrščanje pogleda na vrstice in stolpce, ki se imenuje *GridLayout* [22]. Deluje na enak način, kot bi gradili šahovnico. Pred dodajanjem elementov moramo napisati definicijo, kjer definiramo vrstice z višino in stolpce s širino.

Spodnja koda nazorno prikazuje, kako lahko naredimo mrežo z dvema vrsticama in dvema stolpcema.

```
<Grid ColumnSpacing="3" RowSpacing="3">
 <Grid.RowDefinitions>
  <RowDefinition Height="*"></RowDefinition>
  <RowDefinition Height="*"></RowDefinition>
 </Grid.RowDefinitions>
 <Grid.ColumnDefinitions>
  <ColumnDefinition Width="1*"></ColumnDefinition>
  <ColumnDefinition Width="2*"></ColumnDefinition>
```

```
  </Grid.ColumnDefinitions>
  <BoxView Grid.Row="0" Grid.Column="0" BackgroundColor="Red"></BoxView>
  <BoxView Grid.Row="0" Grid.Column="1" BackgroundColor="Yellow"></BoxView>
  <BoxView Grid.Row="1" Grid.Column="0" Grid.ColumnSpan="2" BackgroundColor="Blue"></BoxView>
</Grid>
```

V kodi smo nastavili več parametrov. Najprej smo nastavili razmik med vrsticami in stolpci v velikosti treh enot (*Spacing*). Višina vrstic je nastavljena na *, kar pomeni, da se bodo vrstice avtomatsko porazdelile, širina stolpcev pa je nastavljena v razmerju 1 : 2. Nastavimo lahko tudi razpon, kjer lahko določen element zasede več stolpcev ali vrstic hkrati. *GridLayout* je predvsem značilen za aplikacije *Windows Phone*. Rezultat zgornje kode za platformi *iOS* in *Windows Phone* je prikazan na sliki 2.5.

Slika 2.5. GridLayout na napravah iOS (levo) in Windows Phone (desno).

2.3.2.5 Pogled vsebine

ContentView je najosnovnejši tip razvrščevalnika, saj lahko vsebuje le enega otroka [23]. Uporablja se predvsem pri kreiranju po meri narejenih kontrol, kjer dedujemo iz tega razreda, ali pa za nastavljanje zapolnitve (*Padding)* elementom, ki tega ne podpirajo.

2.3.2.6 Razvrščanje z drsniki

ScrollView omogoča pomikanje po zaslonu. Največkrat ga uporabimo v navezi s *StackLayoutom*, kjer je *StackLayout* otrok *ScrollViewa*. Pomikanje je možno v primeru, če vsebina zaseda več prostora, kot ga je na voljo na vidnem polju [24].

2.3.2.7 Okvir

Razvrščevalnik *Frame* je na prvi pogled glede funkcionalnosti podoben kot *ContentView*, saj lahko vsebuje samo en element. Prikazuje pravokotni rob okoli vsebine, hkrati ima privzeto še 20 enot zapolnitve [25].

2.3.3 Pogledi

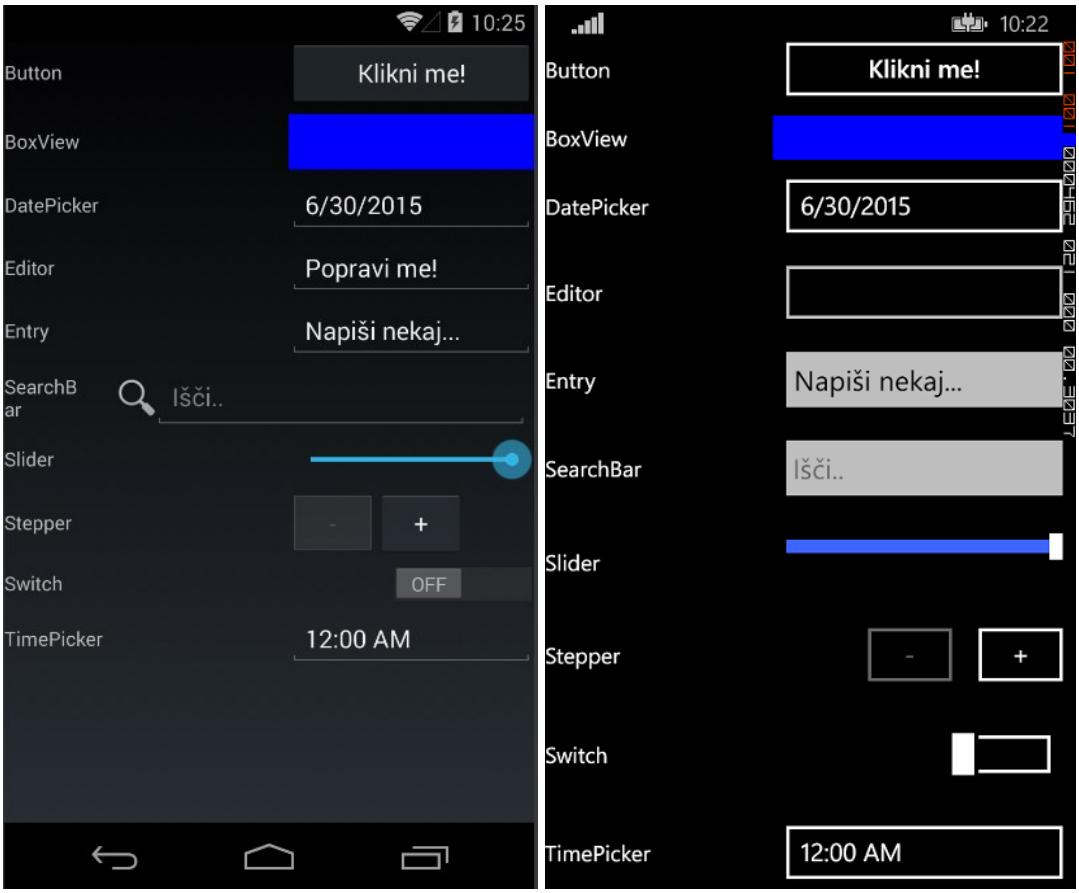
Pogledi predstavljajo vizualne elemente znotraj *Xamarin.Forms*. Definiranih je 19, ki dedujejo od razreda *View*. Večina predstavlja podatke iz ozadja, ki so lahko različnih tipov [26].

Ime pogleda	Tip podatka	Opis
Indikator aktivnosti (ActivityIndicator)	Boolean	Prikazuje trenutno zasedenost aplikacije
Pogled škatla (BoxView)		Omogoča risanje pravokotnikov
Gumb (*Button*)		Gumb
Izbirnik datuma (*DatePicker*)	DateTime	Prikazovalnik datuma
Urejevalnik (*Editor*)	String	Urejevalnik večvrstičnega besedila
Vnosno polje (*Entry*)	String	Urejevalnik enovrstičnega besedila
Slika (*Image*)	Image	Prikazovalnik slik
Oznaka (*Label*)	String	Prikazovalnik besedila
Seznamsko polje (*ListView*)	Object	Prikazovalnik kolekcije podatkov v obliki seznama
Pogled OpenGL (*OpenGLView*)		Prikazovalnik vsebine OpenGL
Izbirnik (*Picker*)	Object	Spustni seznam

Vrstica napredka (*ProgressBar*)	Decimal	Prikazovalnik vrstice napredka
Iskalno okno (*SearchBar*)		Iskalna vrstica
Drsnik (*Slider*)	Integer / Decimal	Drsnik
Koračnik (Stepper)	Integer / Decimal	Omogoča večanje ali manjšanje vrednosti številke
Stikalo (Switch)	Boolean	Stikalo za spreminjanje logične vrednosti
Tabelarični pogled (TableView)	Cell	Tabela, kjer se prikazujejo vrstice tipa *Cell*
Izbirnik časa (TimePicker)	TimeSpan	Prikazovalnik časa
Spletni pogled (WebView)	HTML	Prikazovalnik spletnih strani

Tabela 1. Opis pogledov.

Določeni elementi na platformah *Android* in *Windows Phone* so prikazani na sliki 2.6, kjer lahko vidimo, da se vsi preslikajo v izvorne elemente.



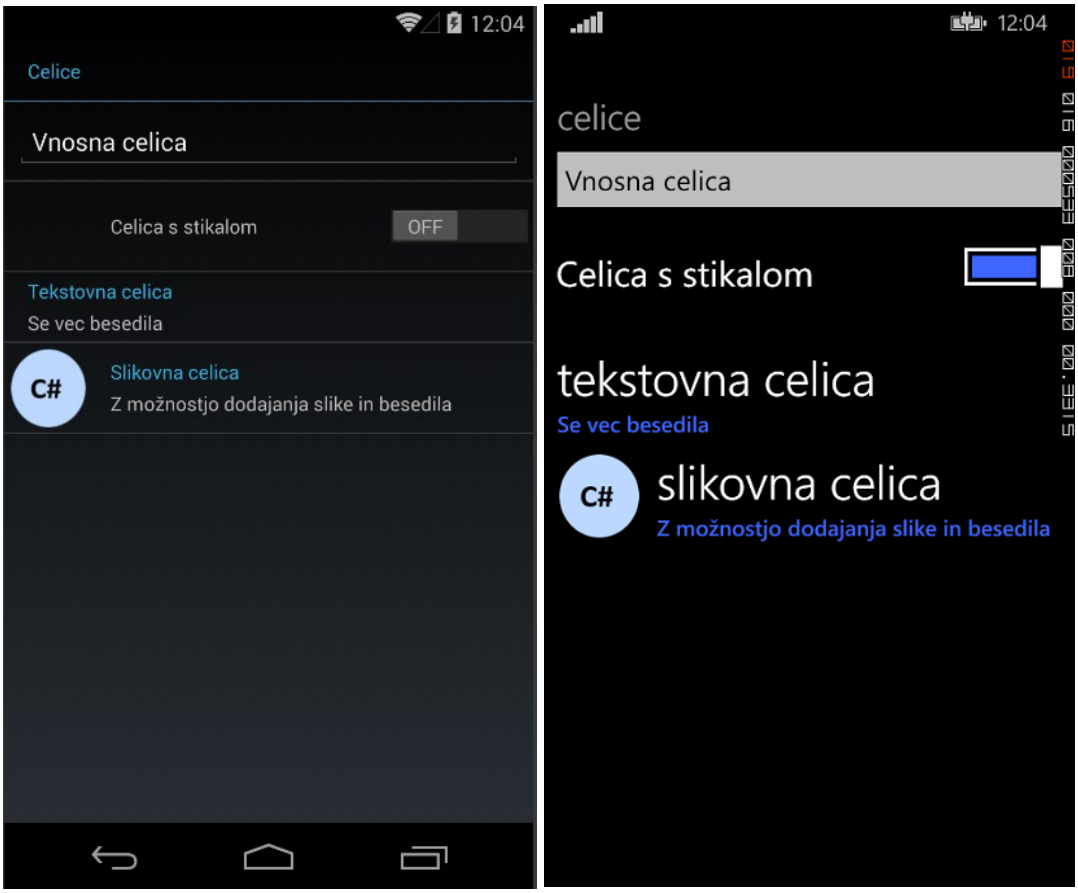
Slika 2.6. Pogledi na napravah Android (levo) in Windows Phone (desno).

2.3.4 Celice

Celica, za razliko od ostalih gradnikov, ni vizualni element – predstavlja zgolj predlogo za njihovo kreiranje. Uporabljamo jih v kombinaciji z *ListView* ali *TableView*. Poznamo štiri vrste celic. Te so [27]:

- vnosna celica (*EntryCell*) – celica z oznako in enovrstičnim vnosnim poljem,
- celica s stikalom (*SwitchCell*) – celica z oznako in s stikalom za spreminjanje logične vrednosti,
- tekstovna celica (*TextCell*) – celica z dvema oznakama (glava in trup),
- slikovna celica (*ImageCell*) – tekstovna celica z možnostjo dodajanja slike.

Primer uporabe različnih tipov celic znotraj *TableView* na platformah *Android* in *Windows Phone* je prikazan na sliki 2.7.



Slika 2.7. Prikaz celic na napravah Android (levo) in Windows Phone (desno).

2.4 Slogi

Sedaj, ko že poznamo nekatere dobre lastnosti jezika *XAML*, lahko gremo še dlje. Če želimo razviti uporabnikom prijazen, moderen in privlačen uporabniški vmesnik, bomo imeli veliko opravka s slogi. Poudarek pri definiranju slogov je seveda znotraj datoteke *XAML*, vendar je vse to možno opraviti tudi znotraj kode – kar pa je seveda teže berljivo. Znotraj *XAML* lahko definiramo že vnaprej pripravljene sloge, jih dinamično spreminjamo, in namesto da bomo določenemu sklopu enakih objektov vsakemu posebej definirali neke lastnosti, je to možno narediti samo enkrat – znotraj slovarja virov (*Resource Dictionary*). Če želimo spreminjati sloge glede na različne dogodke ali lastnosti elementov, pa so sedaj na voljo še sprožilci (*Trigger*).

2.4.1 Upravljanje s slogi

Sloge bomo najprej definirali znotraj slovarja. Najprej nastavimo lastnost *TargetType*, ki nam pove, na kakšen tip elementa bo vplival, znotraj tega pa lahko dodamo poljubno število lastnosti. Znotraj *XAML* je možno tudi dedovati sloge, pri čemer moramo staršu definirati ime prek lastnosti *x:Key*, otroku pa lastnost *BasedOn*. Ni nujno, da lastnost vpliva zgolj na en tip elementa, lahko vpliva na vse vizualne elemente hkrati, v tem primeru nastavimo vrednost na *View*.

Denimo, da imamo več oznak in vnosnih polj, kjer bomo definirali tri sloge. Pri tem bosta sloga za oznake in vnosna polja dedovala od prvega.

```
<ContentPage.Resources>
 <ResourceDictionary>
  <Style TargetType="View" x:Key="baseStyle">
   <Setter Property="HorizontalOptions" Value="CenterAndExpand"></Setter>
  </Style>
  <Style TargetType="Label" BasedOn="{StaticResource baseStyle}">
   <Setter Property="FontAttributes" Value="Bold"></Setter>
   <Setter Property="TextColor" Value="Accent"></Setter>
  </Style>
  <Style TargetType="Entry" BasedOn="{StaticResource baseStyle}">
   <Setter Property="Placeholder" Value="Vnesite vrednost"></Setter>
   <Setter Property="TextColor" Value="Accent"></Setter>
  </Style>
 </ResourceDictionary>
</ContentPage.Resources>
<StackLayout>
 <StackLayout.Children>
  <Label Text="Ime"></Label>
  <Entry></Entry>
```

```
    <Label Text="Priimek"></Label>
    <Entry></Entry>
    <Label Text="Email"></Label>
    <Entry></Entry>
    <Button Text="Prijavi se!"></Button>
  </StackLayout.Children>
</StackLayout>
```

Če želimo sloge spreminjati med delovanjem, moramo namesto statičnega uporabiti dinamični klic. Sloge nastavimo na enak način, le pri sklicevanju na njih uporabimo vrednost *DynamicResource.* Bistvena razlika med delovanji je v tem, da se pri statičnem načinu vir nastavi in razreši med nalaganjem in bodo kakršnekoli spremembe kasneje ignorirane. Pri dinamičnem načinu pa je ravno nasprotno, saj se bo lastnost pridobila le, ko se objekt vpraša za vrednost in kakršnakoli posodobitev bo kasneje upoštevana [28]. Seveda je priporočljivo uporabljati statične vire, saj se kličejo le enkrat in so manj procesno požrešni.

Ob kliku na gumb s pomočjo refleksije pridobimo vse definirane barve iz sistema in spremenimo barvo teksta z naključnim elementom te liste. Predpogoj, da bo koda delovala, je, da uporabimo *DynamicResource*.

Hierarhija znotraj datoteke *XAML* je naslednja:

```
<ContentPage.Resources>
  <ResourceDictionary>
    <Style TargetType="Button" x:Key="button_Style1">
      <Setter Property="TextColor" Value="Blue"></Setter>
    </Style>
  </ResourceDictionary>
</ContentPage.Resources>
<StackLayout>
  <StackLayout.Children>
    <Button Style="{DynamicResource button_Style1}" Text="Klikni za spremembo stila" Clicked="Button1_Click"
x:Name="button1"></Button>
  </StackLayout.Children>
</StackLayout>
```

V kodi pa definiramo dogodek na naslednji način:

```
void Button1_Click(object sender, EventArgs args)
{
   var predefinedColors = typeof (Color).GetTypeInfo().DeclaredFields.Where(el => el.FieldType == typeof (Color)).Select(el
=> el.GetValue(null)).ToList();
   var rand = new Random().Next(0, predefinedColors.Count - 1);

   var style = new Style(typeof (Button));
   var setter = new Setter() {Property = Button.TextColorProperty, Value = predefinedColors[rand]};
   style.Setters.Add(setter);

   Resources["button_Style1"] = style;
}
```

2.4.2 Sprožilci

Sprožilci nam omogočajo spreminjanje videza glede na spremembe dogodkov ali lastnosti elementa. Poznamo jih več vrst [29]:

- lastnostni sprožilec (*Property Trigger*),
- podatkovni sprožilec (*Data Trigger*),
- dogodkovni sprožilec (*Event Trigger*),
- več sprožilec (*Multi Trigger*).

2.4.2.1 Lastnostni sprožilec

Lastnostni sprožilci so uporabni, ko želimo spremeniti vrednost že obstoječih lastnosti elementov. Deklariramo jih lahko na nivoju slovarja virov ali pa znotraj elementa. Če želimo spremeniti barvo ozadja v gumbih, ko so fokusirani, naredimo to na naslednji način:

```
<ContentPage.Resources>
 <ResourceDictionary>
  <Style TargetType="Button">
   <Style.Triggers>
    <Trigger TargetType="Button" Property="IsFocused" Value="True">
     <Setter Property="TextColor" Value="Blue"></Setter>
    </Trigger>
   </Style.Triggers>
  </Style>
 </ResourceDictionary>
</ContentPage.Resources>
```

2.4.2.2 Podatkovni sprožilci

Podatkovni sprožilci se uporabljajo v povezavi s podatkovno vezavo (*Data Binding*). Če želimo, da je lastnost elementa odvisna od vrednosti nekega drugega elementa, bo ta vrsta sprožilca idealna za uporabo. Uporabljamo jih predvsem za validacijo podatkov.

2.4.2.3 Dogodkovni sprožilec

Zastavi se nam vprašanje, kako ukrepati v primeru, ko želimo nastaviti določeno lastnost ob sprožitvi dogodka. To nam sedaj omogočajo dogodkovni sprožilci. Znotraj kode moramo definirati razred, ki deduje od razreda *TriggerAction*, kjer prepišemo metodo *Invoke*.

```
public class EventTriggerExample : TriggerAction<Button>
{
   protected override void Invoke(Button btn)
   {
      btn.Text = "To je dogodkovni sprožilec!";
   }
}
```

2.4.2.4 Več sprožilec

Več sprožilec vsebuje lastnosti lastnostnega in podatkovnega sprožilca. Od njiju se razlikuje v tem, da omogoča definiranje več pogojev hkrati za izvedbo sprožilca.

2.5 Prikazovalnik po meri

Ker je *Xamarin.Forms* še relativno mlado okolje, se nam velikokrat v času razvoja zgodi, da določene lastnosti na elementih še niso podprte. Ker vnosno polje trenutno še ne podpira nastavljanja barve besedila, imamo možnost vse te procese povoziti in definirati svoje prikazovalnike za vsako platformo posebej. Kaj torej so prikazovalniki po meri? Prikazovalniki po meri so most med *Xamarin.Forms* in specifičnimi knjižnicami za platforme *Xamarin* – *Xamarin.Android*, *Xamarin.iOS* kot tudi *Windows Phone SDK*. Kontrole *Xamarin.Forms* so narisane na grafičnem vmesniku z uporabo dveh primernih komponent: elementov in prikazovalnikov [30]. S tem je razvijalcem omogočena popolna kontrola nad izrisovanjem elementov, saj lahko relativno enostavno ustvarimo elemente s svojimi lastnostmi.

Koraki, potrebni za realizacijo prikazovalnikov po meri, so naslednji:

- definirati moramo svoj razred z imenom elementa, ki implementira lastnosti elementa *Xamarin.Forms*,
- znotraj njega nastavimo *BindableProperty*, s katerim bomo lahko nastavili lastnosti iz kode ali *XAML*,
- v platformah, kjer bomo te prikazovalnike uporabili, moramo definirati prikazovalnik po meri, ki bo implementiral prikazovalnik elementa *Xamarin.Forms* in povozimo nastavitve ob kreaciji elementa.

V našem primeru bomo ustvarili element, ki bo vseboval oznako s potrditvenim poljem za platformo *Windows Phone*, kjer bo barva oznake odvisna od logične vrednosti potrditvenega polja.

Najprej ustvarimo razred *CheckBoxLabel* znotraj skupne kode. Definiramo mu tri polja, ki bodo dostopna iz jezika *XAML*. Hkrati moramo biti pozorni na to, da bo razred dedoval iz razreda *View*, saj mu bomo tako lahko nastavili kakršenkoli pogled – v našem primeru *StackPanel*.

```
public class CheckBoxLabel : Xamarin.Forms.View
{
  public static readonly BindableProperty TextProperty = BindableProperty.Create<CheckBoxLabel, string>(p => p.Text, string.Empty);
  public static readonly BindableProperty CheckedForegroundProperty = BindableProperty.Create<CheckBoxLabel, Color>(p => p.CheckedForeground, Color.Default);
  public static readonly BindableProperty UnCheckedForegroundProperty = BindableProperty.Create<CheckBoxLabel, Color>(p => p.UnCheckedForeground, Color.Default);

  public Color CheckedForeground
  {
    get { return (Color)GetValue(CheckedForegroundProperty); }
    set { SetValue(CheckedForegroundProperty, value); }
  }
  public Color UnCheckedForeground
  {
    get { return (Color)GetValue(UnCheckedForegroundProperty); }
    set { SetValue(UnCheckedForegroundProperty, value); }
  }
  public string Text
  {
    get { return (string)GetValue(TextProperty); }
    set { SetValue(TextProperty, value); }
  }
}
```

Naslednji korak, ki sledi, je definicija prikazovalnika znotraj specifične platforme. Tukaj se bomo osredotočili samo na *Windows Phone*. Razred mora vsebovati lastnost *assembly* z vrednostjo razreda in prikazovalnika:

```
[assembly: ExportRenderer(typeof(CheckBoxLabel), typeof(CheckBoxLabelRenderer))]
```

Sledi še implementacija ostalih metod, kjer kreiramo samo izvorne elemente, odvisne od platforme.

```
public class CheckBoxLabelRenderer : ViewRenderer<CheckBoxLabel, System.Windows.Controls.Grid>
{
  protected override void OnElementChanged(ElementChangedEventArgs<CheckBoxLabel> e)
  {
    base.OnElementChanged(e);

    if (e.OldElement == null)
    {
      var xamlData = Element;
      var grid = new System.Windows.Controls.Grid();
      grid.ColumnDefinitions.Add(new System.Windows.Controls.ColumnDefinition());
      grid.ColumnDefinitions.Add(new System.Windows.Controls.ColumnDefinition());
      var textBlock = new System.Windows.Controls.TextBlock()
      {
        Text = xamlData.Text,
        FontSize = 30,
        Foreground = ToColor(xamlData.UnCheckedForeground),
        VerticalAlignment = VerticalAlignment.Center,
        HorizontalAlignment = HorizontalAlignment.Left
      };
      var checkBox = new System.Windows.Controls.CheckBox() { HorizontalAlignment = HorizontalAlignment.Left };

      checkBox.Checked += delegate { textBlock.Foreground = ToColor(xamlData.CheckedForeground); };
      checkBox.Unchecked += delegate { textBlock.Foreground = ToColor(xamlData.UnCheckedForeground); };

      System.Windows.Controls.Grid.SetColumn(textBlock, 0);
      grid.Children.Add(textBlock);
      System.Windows.Controls.Grid.SetColumn(checkBox, 1);
      grid.Children.Add(checkBox);

      this.SetNativeControl(grid);
    }
  }

  public SolidColorBrush ToColor(Xamarin.Forms.Color color)
  {
```

```
        var converter = new ColorConverter();
        var fill = (SolidColorBrush)converter.Convert(color, null, null, null);
        return fill;
    }
}
```

Znotraj metode *OnElementChanged* smo definirali pogled za aplikacijo *Windows Phone*, kjer smo v *Grid* dodali oznako in potrditveno polje. Nanj smo vezali tudi dogodka, kjer bo barva oznake odvisna od logične vrednosti potrditvenega polja.

Preostane nam samo še definiranje datoteke *XAML*, kjer dodamo lastnost *xmlns:local*, ki bo kazala na pot zbirnika znotraj projekta.

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
        xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
        xmlns:local="clr-namespace:App7.CustomRenderer;assembly=App7"
        x:Class="App7.View.CustomRenderXaml">
  <StackLayout HorizontalOptions="FillAndExpand">
    <StackLayout.Children>
      <local:CheckBoxLabel Text="Primer 1" CheckedForeground="Blue"
UnCheckedForeground="Yellow"></local:CheckBoxLabel>
      <local:CheckBoxLabel Text="Primer 2" CheckedForeground="Red"
UnCheckedForeground="Accent"></local:CheckBoxLabel>
    </StackLayout.Children>
  </StackLayout>
</ContentPage>
```

S tem smo uspešno ustvarili prikazovalnik po meri. Če bi želeli enako storiti še za naprave *Android* in *iOS*, bi v drugem koraku morali ustvariti podobne datoteke še v projektih drugih platform.

2.6 Deljenje kode med platformami

Ena izmed večjih polemik med razvijalci, ki razvijajo aplikacije za več operacijskih sistemov hkrati, je deljenje kode med platformami. Kot primer lahko navedemo uporabo zemljevidov *Xamarin*, ki uporablja izvorne zemljevide *Google Maps* za naprave *Android*, *Apple Maps* za *iOS* in *Bing Maps* za *Windows Phone*. Tukaj so morali razvijalci pri *Xamarinu* vzeti skupne funkcionalnosti vseh zemljevidov, zatorej pri vseh kontrolah niso podprte vse funkcionalnosti. Podobni problemi se nam lahko zgodijo pri ostalih funkcionalnostih, kot je npr. kamera ali vmesnik *bluetooth*. V tem poglavju bomo opisali princip injiciranja odvisnosti (»Dependency Injection«) in storitev odvisnosti (»Dependency Service«), s katerima nam *Xamarin* omogoča deljenje kode med platformami.

2.6.1 Injiciranje odvisnosti

Injiciranje odvisnosti je eden izmed mnogih vzorcev, ki se uporabljajo pri razvijanju programske opreme. Je vzorec, ki demonstrira, kako ustvariti ohlapno povezane razrede [31]. Z drugimi besedami to pomeni, da jih objektom, namesto da bi sami ustvarjali instance, mi podamo prek parametrov. Ampak zakaj je to pomembno? Problemi nastajajo predvsem pri testiranju aplikacije, saj v nasprotnem primeru ne moramo ustvarjati »ponarejenih« testnih objektov. Prednosti pri uporabi takšnega načina so naslednje [32]:

- injiciranje odvisnosti omogoča odjemalcu odstraniti vse znanje o konkretnem izvajanju, ki ga potrebuje za uporabo,
- koda je bolj neodvisna, kar nam omogoča lažje testiranje,
- omogoča neodvisno razvijanje, pri katerem bodo razvijalci potrebovali le znanje o vmesniku,
- zmanjšuje povezanost med razredom in njegovo odvisnostjo.

Odvisnosti so lahko injiciranje v objekte na tri načine:

- z injiciranjem konstruktorja,
- z injiciranjem metode,
- z injiciranjem lastnosti.

V primeru razvijanja aplikacij v *Xamarin.Forms* to pomeni, da nam omogoča pisanje kode za specifične funkcije za platforme, kot so prikazovalniki po meri, ravnanje z datotekami, servisi za ozadja in senzorji [33].

2.6.2 Storitve odvisnosti

V *Xamarin.Forms* imamo podprto izvajanje injiciranja odvisnosti s strani razreda *DependencyService*. S pomočjo vmesnika nam omogoča dostop do vseh funkcionalnosti platform *Android*, *iOS* in *Windows Phone SDK* [34].

To storimo v treh korakih:

- najprej definiramo vmesnik v skupni kodi,
- znotraj vseh platform ustvarimo razred, ki implementira ta vmesnik, in s tem izvršimo registracijo,
- v skupni kodi prek statičnega klica *DependencyService.Get<T>* izvršimo klic na omenjeni vmesnik.

V našem primeru bomo ustvarili vmesnik, ki bo aplikacijam za SMS-sporočila posredoval telefonsko številko in sporočilo. Primer bo narejen za naprave *Android* in *Windows Phone*.

Na začetku ustvarimo vmesnik *IMessage* znotraj skupne kode:

```
public interface IMessage
{
   void SendMessage(string to,string message);
}
```

Sedaj moramo ustvariti še dva ločena razreda *Message_Android* in *Message_WinPhone*, ki bosta implementirala metodo *SendMessage*. Razreda se morata nahajati v projektu naprave, ki jo bosta podpirala. Hkrati jima moramo dodati lastnost za *Dependency*, katere vrednost bo tip razreda :

```
[assembly: Dependency(typeof(/*Ime razreda*/))]
```

Implementacija razredov *Message_Android* in *Message_WinPhone*:

```
class Message_Android : IMessage
{
   public Message_Android() { }
   public void SendMessage(string to, string message)
   {
      var smsUri = Uri.Parse(string.Format("smsto:{0}", to));
      var smsIntent = new Intent(Intent.ActionSendto, smsUri);
      smsIntent.PutExtra("sms_body", message);
      var ac = new Activity();
      ac.StartActivity(smsIntent);
```

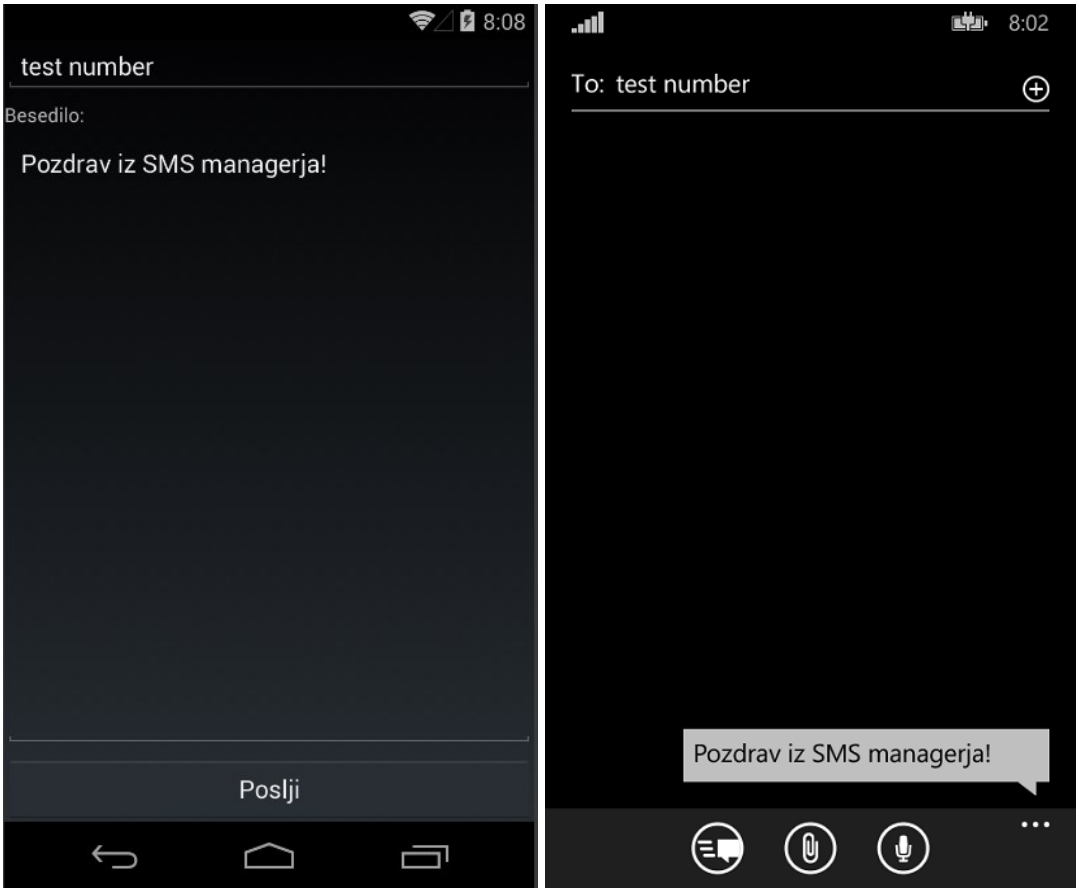
```
    }
}

class Message_WinPhone : IMessage
{
    public Message_WinPhone() { }
    public void SendMessage(string to, string message)
    {
        SmsComposeTask smsComposeTask = new SmsComposeTask();
        smsComposeTask.To = to;
        smsComposeTask.Body = message;
        smsComposeTask.Show();
    }
}
```

Ker bomo pri *Androidu* dostopali do SMS-vmesnika, moramo dodati še pravice v datoteko *AndroidManifest.xml*:

```
<uses-permission android:name="android.permission.SEND_SMS" />
```

Kot zadnji korak sledi še samo klicanje vmesnika prek *DependencyService*. To storimo kar prek klika na gumb (spremenljivki *To* in *Message* sta niza):

```
return new Command(() =>
{
    DependencyService.Get<IMessage>().SendMessage(To,Message);
    //pobrišimo vrednosti vnosnih polj
    To = string.Empty;
    Message = string.Empty;
});
```

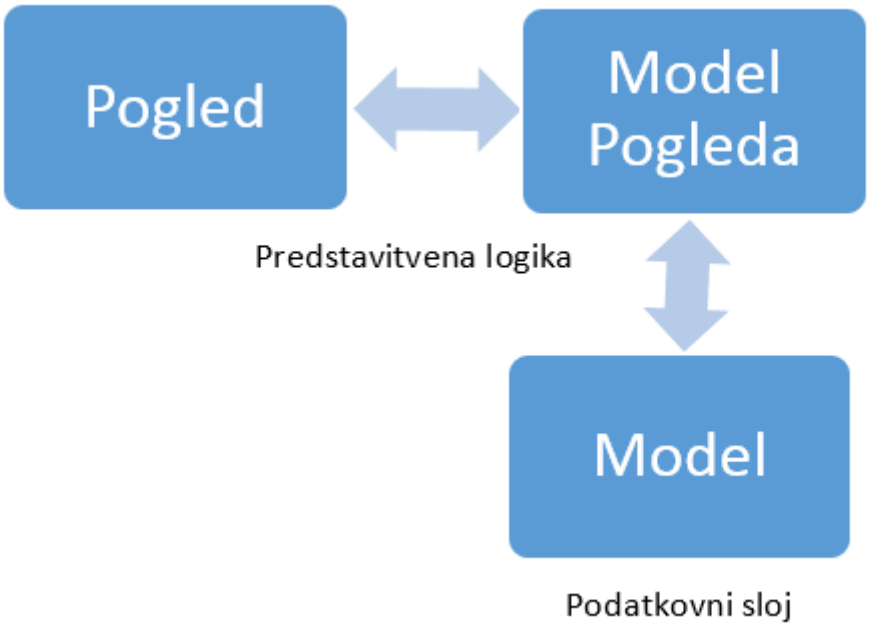


Slika 2.8. Pošiljanje sporočila na napravah Android (levo) in Windows Phone (desno).

Rezultat za obe platformi je prikazan na sliki 2.8. Na napravi *Android* je prikazan grafični vmesnik, ki od nas prejme telefonsko številko in besedilo sporočila. Po kliku na gumb *Poslji* se nam odpre aplikacija za pošiljanje SMS-sporočil, kot je vidno na napravi *Windows Phone*.

3 Arhitekturni slog model-pogled-model pogleda

Arhitekturni slogi se v računalništvu uporabljajo že od samega začetka. Navezujejo se na visokonivojsko strukturo programskega sistema, disciplino ustvarjanja takih struktur in njihovo dokumentacijo [35]. Njihov namen uporabe je predvsem vnovična uporabljivost v prihodnosti in prilagodljivost. Kot je že v ustvarjanju aplikacij z grafičnimi vmesniki samoumevno, da bomo ločevali logiko programa od uporabniškega vmesnika, so tudi pri arhitekturnih slogih napisana takšna pravila. Uporablja se jih veliko, v našem primeru pa se bomo osredotočili na slog model-pogled-model pogleda (»MVVM«), saj je bil ustvarjen specifično za notacijo *XAML*. Ta slog nam pomaga ločiti poslovno in predstavitveno logiko aplikacije od uporabniškega vmesnika, saj nam ohranjanje takšne razporeditve omogoča hitrejše razreševanje napak, testiranja, vzdrževanja in razvijanja v prihodnje [36].



Slika 3.1. Razredi in njihove interakcije.

Kot je razvidno iz slike 3.1, se slog MVVM deli na naslednje plasti:

- model (»Model«),
- pogled (»View«),
- model pogleda (»View-Model«).

Projekt se nato razdeli na tri dele, ki ustrezajo zgoraj omenjenim trem plastem. Velja tudi pravilo, da ima vsaka datoteka dodano končnico mape, kamor spada (npr. razred Osebe, ki pripada mapi *ViewModel*, se bo imenoval *OsebeViewModel*).

Pri pisanju kode po vzorcu model-pogled-model pogleda hitro ugotovimo, da se nam del kode neprestano ponavlja. V tem primeru si lahko pomagamo z že obstoječimi okvirji, s čimer si lahko prihranimo veliko časa:

- *MVVM Light Toolkit*,
- *MvvmCross*,
- *ReactiveUI*,
- *FreshMvvm*.

3.1 Pogled

Namen pogleda je predstaviti uporabniški vmesnik s pomočjo jezika *XAML*. Odgovoren je za zapis celotne strukture, kjer poskusimo čim manj uporabljati kodo. Je v interakciji z model pogleda, in sicer prek podatkovne vezave (»Data Binding«) ter ukazov (»Commands«).

Na spodnjem primeru ustvarimo obrazec za pošiljanje sporočila razvijalcem aplikacije. Kot vidimo, smo pri vnosnih poljih na lastnost *Text* vezali pripadajočo podatkovno vezavo, na gumbu pa smo napolnili argument *Command*. Hkrati smo ustvarili še prikazovalnik po meri (»Custom renderer«), saj pri trenutnih nastavitvah ni možno nastaviti barve teksta v vnosnem polju. Problem je nastal na aplikaciji *Windows Phone*, saj ko je vnosno polje fokusirano, je besedilo teksta enako ozadju polja.

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
        xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
        x:Class="Application.View.MessageView"
        xmlns:local="clr-namespace:Application.CustomRenderer;assembly=Application">
 <ContentPage.Resources>
  <ResourceDictionary>
   <Style TargetType="Entry">
```

```
      <Setter Property="HorizontalOptions" Value="FillAndExpand"/>
    </Style>
    <Style TargetType="Label">
      <Setter Property="VerticalOptions" Value="Center"/>
    </Style>
  </ResourceDictionary>
 </ContentPage.Resources>
 <StackLayout>
  <StackLayout.Children>
   <ContentView Padding="10,5,5,0">
    <StackLayout>
     <StackLayout.Children>
      <StackLayout Orientation="Horizontal">
       <StackLayout.Children>
        <Label Text="Ime" WidthRequest="100"></Label>
        <Entry Text="{Binding Ime}"></Entry>
       </StackLayout.Children>
      </StackLayout>
      <StackLayout Orientation="Horizontal">
       <StackLayout.Children>
        <Label Text="Email" WidthRequest="100"></Label>
        <Entry Text="{Binding Email}"></Entry>
       </StackLayout.Children>
      </StackLayout>
      <Label Text="Vprašanje"></Label>
     </StackLayout.Children>
    </StackLayout>
   </ContentView>
   <local:MyEditor Text="{Binding Vprasanje}" VerticalOptions="FillAndExpand" Foreground="White"></local:MyEditor>
   <Button Text="Pošlji" Command="{Binding SendMessage}" BackgroundColor="Accent"></Button>
  </StackLayout.Children>
 </StackLayout>
</ContentPage>
```

3.2 Model pogleda

Model pogled nam služi kot povezava med grafičnim vmesnikom in interakcijami z njim. Ne vsebuje neposredne reference na pogled in nima vpogleda v grafični vmesnik. V idealni kodi je lahko spisan neodvisno od platforme, zato ga lahko v tej obliki kasneje uporabimo še na drugih platformah, kot sta *Windows Presentation Form* ali *Silverlight*. Toda na kakšen način uredimo to povezavo? Sprva moramo na pogledu definirati *BindingContext*, s katerim bomo pogledu povedali, na kateri razred se mora sklicevati. A bistvo se skriva v vmesniku *INotifyPropertyChanged*. Da bo podatkovna povezava delovala, mora model pogleda

implementirati sporočilni protokol, ki bo pošiljal signale, ko se lastnost v modelu pogleda spremeni [37]. Za primer iz podpoglavja 3.1 še ustrezno definiramo model pogleda s sporočilnim vmesnikom. Vrednosti lastnosti *Binding*, ki smo jih definirali v pogledu, morajo biti enake spremenljivkam znotraj modela pogleda.

```
public class MessageViewModel : INotifyPropertyChanged
{
   public event PropertyChangedEventHandler PropertyChanged;
   protected void OnPropertyChanged(string propertyName)
   {
      PropertyChangedEventHandler handler = PropertyChanged;
      if (handler != null)
      {
         handler(this, new PropertyChangedEventArgs(propertyName));
      }
   }
   private string _ime;
   private string _vprasanje;
   private string _email;
   public string Ime
   {
      set
      {
         if (_ime != value)
         {            this._ime = value;
            OnPropertyChanged("Ime");
         }
      }
      get { return this._ime; }
   }
   public string Vprasanje
   {
      set
      {
         if (_vprasanje != value)
         {
            this._vprasanje = value;
            OnPropertyChanged("Vprasanje");
         }
      }
      get { return this._vprasanje; }
   }
   public string Email
   {
      set
      {
```

```
            if (_email != value)
            {
                this._email = value;
                OnPropertyChanged("Email");
            }
        }
        get { return this._email; }
    }
    public ICommand SendMessage
    {
        get
        {
            return new Command(() =>
            {
                Helpers.SendMessage(Ime, Email, Vprasanje);
                App.Current.MainPage.DisplayAlert("Informacija", "Sporocilo uspesno poslano!", "Nazaj");
                //pobrišimo vrednosti vnosnih polj
                Ime = string.Empty;
                Email = string.Empty;
                Vprasanje = string.Empty;
            });
        }
    }
}
```

Kaj vse smo morali definirati za pravilno delovanje modela pogleda? Najprej smo razredu implementirali vmesnik *INotifyPropertyChanged.* Metoda *OnPropertyChanged* bo skrbela za pošiljanje signala do pogleda. Sprožila se bo v primeru naslavljanja izbranih spremenljivk, s čimer bomo videli spremembo na uporabniškem vmesniku. V primeru ukaza pa bomo samo vrnili nov ukaz, pri čemer pošljemo sporočilo prek statične metode *SendMessage,* za konec pa uporabnika obvestimo o uspešnosti akcije in ponastavimo vrednosti vnosnih polij na prvotno stanje.

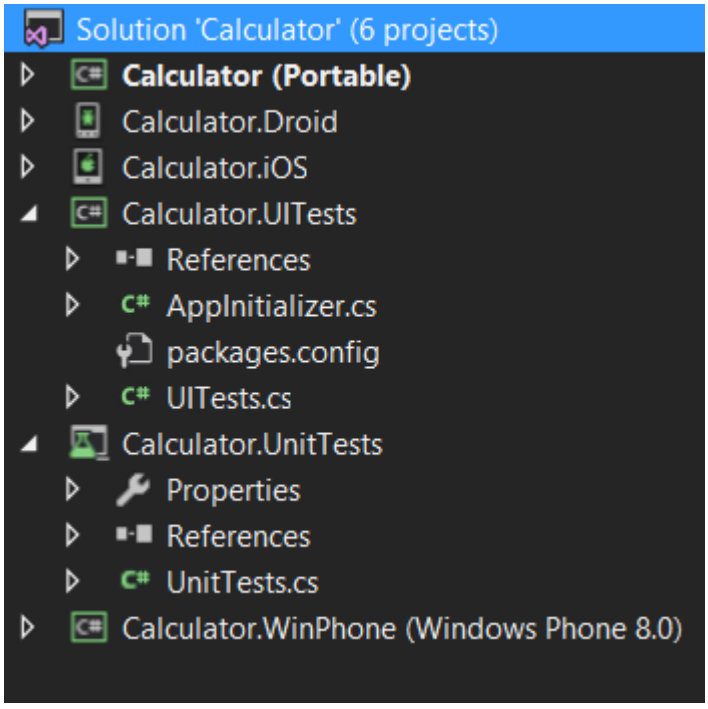
3.3 Model

Model zajema podatkovni sloj aplikacije. Tipično predstavlja model domene za odjemalca za aplikacijo. Pogosto sta model in dostop do sloja podatkov generirana kot del podatkovnega dostopa, kot sta *ADO.NET Entity Framework* ali *WCF Data Services*.

4 Testiranje

Testiranje programske opreme je način preiskave, ki zagotovi zainteresirani skupini informacijo o kakovosti produkta ali storitve, ki se preizkuša [38]. Pri podjetju *Xamarin* so razvili *Xamarin.UITest*, ki deluje na podlagi vmesnika *Calabash*. Je testni vmesnik, ki omogoča avtomatske teste za uporabniške vmesnike, spisane v *NUnit*, ki jih lahko poganjamo na napravah *Android* in *iOS*. Zanaša se na *Xamarin Test Cloud Agent*, kjer poteka komunikacija z napravo prek HTTP-standarda [39]. Testiranje je ključnega pomena za iskanje napak v kodi, saj na takšen način najhitreje poiščemo vse hrošče. A poleg omenjenega testiranja nam je na voljo še privzeto testiranje kode **C#** s pomočjo enot testiranja (»*Unit Testing*«), ki nam je v pomoč pri testiranju skupne kode.

Za primer bomo naredili aplikacijo z imenom *Calculator*, ki bo poleg standardnih projektov vsebovala še projekt za testiranje uporabniškega vmesnika s pomočjo *Xamarin.UITest* in testiranje logike programa s testi enot. Ko obstoječemu projektu *Xamarin.Forms* dodamo še tipa projekta *UITest* in *Unit Test Project*, je drevesna struktura takšna, kot je prikazana na sliki 4.1.



Slika 4.1. Drevesna struktura projekta s testi uporabniškega vmesnika in testi enot.

4.1 Testiranje uporabniškega vmesnika

V naslednjem primeru bomo preverili, če se na uporabniškem vmesniku nahaja gumb za izračun z imenom *btn_Result*. Koda za generiranje *Xamarin.UITest* je naslednja:

```
[TestFixture]
public class UITests
{
   private AndroidApp app;
   private string path = "Calculator.Droid.apk";

   [SetUp]
   public void Start()
   {
      app = ConfigureApp.Android.ApkFile(path).StartApp();
   }

   [Test]
   public void IsResultButtonVisible()
   {
      Func<AppQuery, AppQuery> resultButton = e => e.Id("btn_Result");
      app.WaitForElement(resultButton);
      var btn = app.Query(resultButton).SingleOrDefault();
      Assert.IsNotNull(btn);
   }
}
```

Najprej moramo definirati pot do datoteke *.apk*, ki se nahaja znotraj projekta. Pri testiranju se na začetku izvrši metoda, ki vsebuje lastnost *[SetUp]*, v našem primeru je to *Start*. Znotraj te se bo aplikacija zagnala in nato se bodo pričele izvrševati metode z lastnostjo *[Test]*. Vse, kar smo v metodi zapisali, je, da v spremenljivko *btn* shranimo gumb z imenom *btn_Result* v primeru, če obstaja zgolj en primerek, drugače se zapiše vrednost *null*. Na koncu bo *Assert* iz vmesnika *NUnit* poskrbel za preverjanje rezultata.

4.2 Testiranje enot

Sedaj pokažemo še, kako se izvede test enot. V skupni kodi je definiran statični razred z imenom *Helpers*, ki vsebuje statično metodo *Divide*, ki kot parametra prejme dve števili. V našem primeru se prepričamo, da bo aplikacija vrnila izjemo, ko delimo s številom 0. Najprej dodamo projektu *Calculator.UnitTests* kot referenco projekt *Calculator*, da bo lahko dostopal do razreda *Helpers*.

Metoda za deljenje je spisana na naslednji način:

```
public static decimal Divide(int number1, int number2)
{
   if (number2 == 0)
   {
      throw new Exception("Cannot divide by zero.");
   }
   return (number1 / (decimal)number2);
}
```

Pri testiranju enot pa dodamo testni metodi lastnost *ExpectedException*, saj pričakujemo napako v primeru deljenja s številom 0.

```
[TestMethod]
[ExpectedException(typeof(Exception), "Cannot divide by zero.")]
public void DivideThrowException()
{
   var num1 = 5;
   var num2 = 0;
   var result = Helpers.Divide(num1, num2);
}
```

5 Sklepne ugotovitve

Kot je že v prvem poglavju omenjeno, se čedalje več internetnih vsebin prenaša na mobilne aplikacije, predvsem zaradi porasta števila pametnih telefonov v svetu. Trenutno največ naprav temelji na platformah *Android*, *iOS* in *Windows Phone*. Kot posledica tega se nam ponuja vedno več alternativ za razvoj mobilnih aplikacij, nas pa je še posebej zanimal izdelek podjetja *Xamarin*, imenovan *Xamarin.Forms*. Kot smo prikazali v drugem poglavju, je načrtovanje grafičnega vmesnika zelo enostavno, saj nam *XAML* omogoča, da v zelo kratkem času ustvarimo zelo pregledne vmesnike. Težava je predvsem ta, da smo zaradi odsotnosti grafičnega urejevalnika pri dizajniranju prepuščeni lastni intuiciji, kar lahko predstavlja resno oviro pri izdelavi kompleksnejših grafičnih uporabniških vmesnikov. Na predavanjih *Xamarin HackDays* v Ljubljani smo zasledili, da priporočajo razvoj v *Xamarin.Forms* samo v primeru, ko gre za manjše aplikacije, ki največ delajo s podatki in ne potrebujejo preveč zahtevnega grafičnega vmesnika.

V preteklem obdobju smo z uporabo predstavljenega okolja poskusili ustvariti veliko raznovrstnih projektov in opazili v njem veliko potenciala, predvsem z vidika deljenja kode med vsemi platformami. Za potrebe diplomske naloge smo razvili več aplikacij, da smo lahko zajeli čim večji nabor funkcionalnosti, ki jih podpira *Xamarin.Forms*. Poudariti je treba, da je ta izdelek še relativno mlad in zato vsebuje še veliko hroščev, prav tako pogrešamo več funkcionalnosti. Ravno relativna nedozorelost tovrstnih večplatformnih okolij je glavni razlog, da se veliko ljudi izogiba večplatformnih idej. Kot primer lahko navedemo težave s pomnilnikom pri *Androidu,* kot posledico tega, ko delamo s slikami ali z zemljevidi. V primeru nepravilnega ravnanja z njimi lahko prihaja do prekomerne porabe pomnilnika, tako da moramo včasih sami poskrbeti za osvobajanje elementov, ki niso več v uporabi. A vse to je treba vzeti v zakup, če želimo v najkrajšem možnem času razviti aplikacijo za čim več platform in s tem čim več naprav hkrati. S pomočjo vmesnika *Calabash* [40] so v podjetju Xamarin razvili tudi zelo priročno testiranje uporabniškega vmesnika in treba je omeniti še njihov izdelek *Xamarin Test Cloud* [41], ki nam omogoča testiranje aplikacije z več kot 1000 napravami v oblaku. Trenutna slabost omenjenega izdelka je predvsem prevelika cena, zaradi katere bi si ga manjša podjetja stežka privoščila. Kot pozitivno stran lahko navedem odlično podporo študentom, saj omogočajo brezplačno uporabljanje podjetniških licenc za obdobje enega leta.

5.1 Prednosti in slabosti uporabe Xamarin.Forms

Poglejmo si še nekaj bistvenih prednosti uporabe okolja *Xamarin.Forms*:

- omogoča izgradnjo izvornih aplikacij za tri najpopularnejše mobilne platforme,
- uporabimo lahko vse funkcionalnosti okolja *.NET* in knjižnic za platformi *Android* ali *iOS*,
- *DependencyService* podpira enostaven način dostopa do specifičnih klicev za platformo,
- ustvarimo lahko prikazovalnike po meri,
- aplikacije je možno razviti v arhitekturnem stilu MVVM ali MVC,
- projekt lahko razvijamo v *Visual Studio* ali *Xamarin Studio*,
- vgrajen vmesnik za *SQLite*,
- razvijanje aplikacij za *Google play* in *Apple store*,
- zgrajen na ogrodju odprtokodnega okolja *Mono*, zaradi česar lahko uporabljamo aplikacije na ostalih platformah,
- *Xamarin.Forms* omogoča pisanje iste kode za uporabniški vmesnik za vse platforme,
- razvijanje uporabniškega vmesnika s pomočjo jezika *XAML*, ki omogoča tudi podatkovno vezavo,
- *Xamarin Test Cloud* omogoča avtomatsko testiranje aplikacij,
- hitro rastoča skupnost.

Seveda obstaja tudi več slabosti pri takšnem razvijanju aplikacij :

- aplikacije razvite s *Xamarin.Forms* bodo zavzemale več prostora zaradi *.NET* zbirnikov,
- ne vsebuje uporabniškega urejevalnika za *XAML*,
- ne vsebuje vseh elementov za gradnjo uporabniškega vmesnika za vse platforme,
- visoka cena licenc za razvijalce,
- v primeru posodobitev za *iOS* ali *Android* platforme, bo potrebno počakati na razvijalce v Xamarinu, da jih dodajo v okolje,
- v vsakem primeru je potrebno razumeti arhitekturo *Android*, *iOS* in *Windows Phone* platforme, če želimo podpirati funkcionalnosti,
- počasnost nameščanja aplikacij v primeru aplikacij *Android*,
- težave s prekomerno uporabo pomnilnika,
- razvoj aplikacij *iOS* zahteva operacijski sistem *Mac OS X,*
- nezmožnost uporabe programskega jezika **Java** ali **Objective-C**.

Literatura

[1] (2010) Jolie O'Dell: New Study Shows the Mobile Web Will Rule by 2015 [STATS]. Dostopno na: http://mashable.com/2010/04/13/mobile-web-stats/.

[2] Mobile application development.
Dostopno na: https://en.wikipedia.org/wiki/Mobile_application_development.

[3] Android NDK. Dostopno na: https://developer.android.com/tools/sdk/ndk/index.html.

[4] (2014) Getting started with developing for Windows Phone 8. Dostopno na: https://msdn.microsoft.com/en-us/library/windows/apps/ff402529%28v=vs.105%29.aspx.

[5] An Introduction to Xamarin.Forms. Dostopno na: http://developer.xamarin.com/guides/cross-platform/xamarin-forms/introduction-to-xamarin-forms/.

[6] About Mono. Dostopno na: http://www.mono-project.com/docs/about-mono/.

[7] (2014) Nat Friedman: Announcing Xamarin 3. Dostopno na: https://blog.xamarin.com/announcing-xamarin-3/.

[8] Introduction to Portable Class Libraries. Dostopno na: http://developer.xamarin.com/guides/cross-platform/application_fundamentals/pcl/introduction_to_portable_class_libraries/.

[9] Petzold, C. (2015). XAML vs. code. V D. Musgrave (ur.), Creating Mobile Apps with Xamarin.Forms Preview Edition 2 (Developer Reference) (str. 121–144). Washington: Microsoft Press.

[10] What is XAML?. Dostopno na: https://msdn.microsoft.com/en-us/library/cc295302.aspx.

[11] Petzold, C. (2015). Platform-specific API calls. V D. Musgrave (ur.), Creating Mobile Apps with Xamarin.Forms Preview Edition 2 (Developer Reference) (str. 170–185). Washington: Microsoft Press.

[12] Petzold, C. (2015). Code and XAML in harmony. V D. Musgrave (ur.), Creating Mobile Apps with Xamarin.Forms Preview Edition 2 (Developer Reference) (str. 145–169). Washington: Microsoft Press.

[13] Hermes, D. (2015). Data Access with SQLite and Data Binding. V J. DeWolf (ur.), Xamarin Mobile Application Development: Cross-Platform C# and Xamarin.Forms Fundamentals (str. 297–348). New York: Apress.

[14] Working with Gestures. Dostopno na: http://developer.xamarin.com/guides/cross-platform/xamarin-forms/working-with/gestures/.

[15] Xamarin.Forms Controls Reference. Dostopno na: http://developer.xamarin.com/guides/cross-platform/xamarin-forms/controls/.

[16] Xamarin.Forms Pages. Dostopno na: http://developer.xamarin.com/guides/cross-platform/xamarin-forms/controls/pages/.

[17] Xamarin.Forms.NavigationPage Class. Dostopno na: http://developer.xamarin.com/api/type/Xamarin.Forms.NavigationPage/.

[18] Xamarin.Forms Layouts. Dostopno na: http://developer.xamarin.com/guides/cross-platform/xamarin-forms/controls/layouts/.

[19] Xamarin.Forms.StackLayout Class. Dostopno na: http://developer.xamarin.com/api/type/Xamarin.Forms.StackLayout/.

[20] Petzold, C. (2015). Absolute layout. V D. Musgrave (ur.), Creating Mobile Apps with Xamarin.Forms Preview Edition 2 (Developer Reference) (str. 322–354). Washington: Microsoft Press.

[21] Xamarin.Forms.RelativeLayout Class. Dostopno na: http://developer.xamarin.com/api/type/Xamarin.Forms.RelativeLayout/.

[22] Hermes, D. (2015). UI Design Using Layouts. V J. DeWolf (ur.), Xamarin Mobile Application Development: Cross-Platform C# and Xamarin.Forms Fundamentals (str. 45–104). New York: Apress.

[23] Xamarin.Forms.ContentView Class. Dostopno na: http://developer.xamarin.com/api/type/Xamarin.Forms.ContentView/.

[24] Xamarin.Forms.ScrollView Class. Dostopno na: http://developer.xamarin.com/api/type/Xamarin.Forms.ScrollView/.

[25] Xamarin.Forms.Frame Class. Dostopno na: http://developer.xamarin.com/api/type/Xamarin.Forms.Frame/.

[26] Xamarin.Forms Views. Dostopno na: http://developer.xamarin.com/guides/cross-platform/xamarin-forms/controls/views/.

[27] Xamarin.Forms Cells. Dostopno na: http://developer.xamarin.com/guides/cross-platform/xamarin-forms/controls/cells/.

[28] Petzold, C. (2015). Styles. V D. Musgrave (ur.), Creating Mobile Apps with Xamarin.Forms Preview Edition 2 (Developer Reference) (str. 240–268). Washington: Microsoft Press.

[29] Working with Triggers. Dostopno na: http://developer.xamarin.com/guides/cross-platform/xamarin-forms/working-with/triggers/.

[30] Hermes, D. (2015). Custom Renderers. V J. DeWolf (ur.), Xamarin Mobile Application Development: Cross-Platform C# and Xamarin.Forms Fundamentals (str. 349–366). New York: Apress.

[31] (2011) Alex Culp: The Dependency Injection Design Pattern. Dostopno na: https://msdn.microsoft.com/en-us/library/vstudio/hh323705%28v=vs.100%29.aspx.

[32] Dependency injection. Dostopno na: https://en.wikipedia.org/wiki/Dependency_ injection.

[33] Hermes, D. (2015). Cross-Platform Arhitecture. V J. DeWolf (ur.), Xamarin Mobile Application Development: Cross-Platform C# and Xamarin.Forms Fundamentals (str. 367–386). New York: Apress.

[34] Accessing Native Features via the DependencyService. Dostopno na: http://developer.xamarin.com/guides/cross-platform/xamarin-forms/dependency-service/.

[35] Software architecture. Dostopno na: https://en.wikipedia.org/wiki/Software_architecture.

[36] (2014) Implementing the MVVM Pattern Using the Prism Library 5.0 for WPF. Dostopno na: https://msdn.microsoft.com/en-us/library/gg405484%28v=pandp.40%29.aspx.

[37] Petzold, C. (2015). MVVM. V D. Musgrave (ur.), Creating Mobile Apps with Xamarin.Forms Preview Edition 2 (Developer Reference) (str. 471–514). Washington: Microsoft Press.

[38] Software testing. Dostopno na: https://en.wikipedia.org/wiki/Software_testing.

[39] Introduction to Xamarin.UITest. Dostopno na: http://developer.xamarin.com/guides/ testcloud/uitest/intro-to-uitest/.

[40] Introduction to Calabash. Dostopno na: http://developer.xamarin.com/guides/testcloud/ calabash/introduction-to-calabash/.

[41] Introduction to Xamarin Test Cloud. Dostopno na: http://developer.xamarin.com/guides/testcloud/introduction-to-test-cloud/.